

Isabelle/HOLCF — Higher-Order Logic of Computable Functions

May 23, 2024

Contents

1	Partial orders	3
1.1	Type class for partial orders	3
1.2	Upper bounds	4
1.3	Least upper bounds	5
1.4	Countable chains	6
1.5	Finite chains	7
2	Classes <code>cpo</code> and <code>pcpo</code>	8
2.1	Complete partial orders	8
2.2	Pointed cpos	10
2.3	Chain-finite and flat cpos	10
2.4	Discrete cpos	11
3	Continuity and monotonicity	11
3.1	Definitions	12
3.2	Equivalence of alternate definition	12
3.3	Collection of continuity rules	13
3.4	Continuity of basic functions	13
3.5	Finite chains and flat pcpos	14
4	Admissibility and compactness	14
4.1	Definitions	14
4.2	Admissibility on chain-finite types	15
4.3	Admissibility of special formulae and propagation	15
4.4	Compactness	16
5	Subtypes of <code>pcpos</code>	17
5.1	Proving a subtype is a partial order	17
5.2	Proving a subtype is finite	18
5.3	Proving a subtype is chain-finite	18

5.4	Proving a subtype is complete	18
5.4.1	Continuity of <i>Rep</i> and <i>Abs</i>	19
5.5	Proving subtype elements are compact	19
5.6	Proving a subtype is pointed	20
5.6.1	Strictness of <i>Rep</i> and <i>Abs</i>	20
5.7	Proving a subtype is flat	21
5.8	HOLCF type definition package	21
6	Class instances for the full function space	21
6.1	Full function space is a partial order	21
6.2	Full function space is chain complete	22
6.3	Full function space is pointed	22
6.4	Propagation of monotonicity and continuity	23
7	The cpo of cartesian products	23
7.1	Unit type is a pcpo	24
7.2	Product type is a partial order	24
7.3	Monotonicity of <i>Pair</i> , <i>fst</i> , <i>snd</i>	24
7.4	Product type is a cpo	25
7.5	Product type is pointed	26
7.6	Continuity of <i>Pair</i> , <i>fst</i> , <i>snd</i>	26
7.7	Compactness and chain-finiteness	27
8	The type of continuous functions	28
8.1	Definition of continuous function type	28
8.2	Syntax for continuous lambda abstraction	28
8.3	Continuous function space is pointed	29
8.4	Basic properties of continuous functions	29
8.4.1	Beta-reduction simproc	30
8.5	Continuity of application	30
8.6	Continuity simplification procedure	32
8.7	Miscellaneous	33
8.8	Continuous injection-retraction pairs	33
8.9	Identity and composition	34
8.10	Strictified functions	35
8.11	Continuity of let-bindings	35
9	Continuous deflations and ep-pairs	36
9.1	Continuous deflations	36
9.2	Deflations with finite range	37
9.3	Continuous embedding-projection pairs	38
9.4	Uniqueness of ep-pairs	39
9.5	Composing ep-pairs	39

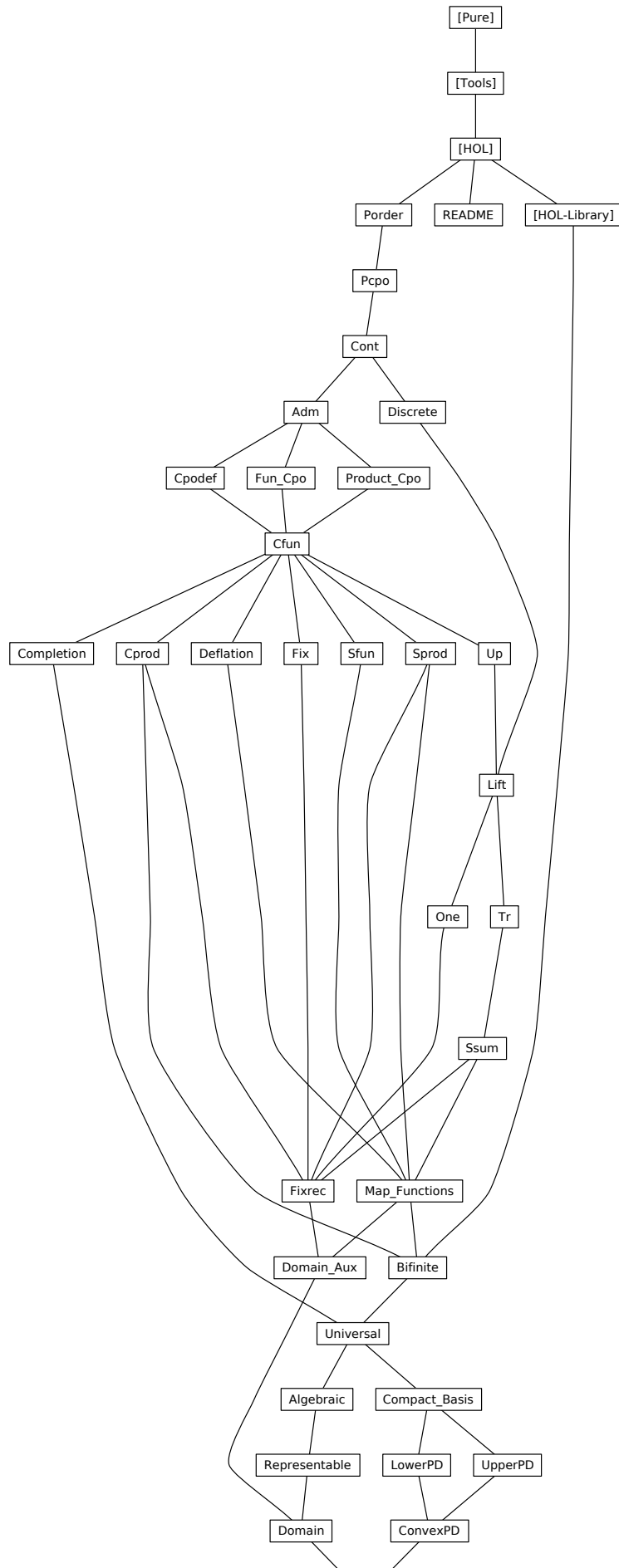
10 The type of strict products	40
10.1 Definition of strict product type	40
10.2 Definitions of constants	40
10.3 Case analysis	41
10.4 Properties of <i>spair</i>	41
10.5 Properties of <i>sfst</i> and <i>ssnd</i>	42
10.6 Compactness	43
10.7 Properties of <i>ssplit</i>	43
10.8 Strict product preserves flatness	44
11 Discrete cpo types	44
11.1 Discrete cpo class instance	44
11.2 <i>undiscr</i>	44
12 The type of lifted values	44
12.1 Definition of new type for lifting	45
12.2 Ordering on lifted cpo	45
12.3 Lifted cpo is a partial order	45
12.4 Lifted cpo is a cpo	45
12.5 Lifted cpo is pointed	46
12.6 Continuity of <i>Iup</i> and <i>Ifup</i>	46
12.7 Continuous versions of constants	46
13 Lifting types of class type to flat pcpo's	48
13.1 Lift as a datatype	48
13.2 Lift is flat	49
13.3 Continuity of <i>case-lift</i>	49
13.4 Further operations	49
14 The type of lifted booleans	50
14.1 Type definition and constructors	50
14.2 Case analysis	51
14.3 Boolean connectives	51
14.4 Rewriting of HOLCF operations to HOL functions	52
14.5 Compactness	53
15 The type of strict sums	53
15.1 Definition of strict sum type	53
15.2 Definitions of constructors	54
15.3 Properties of <i>sinl</i> and <i>sinr</i>	54
15.4 Case analysis	56
15.5 Case analysis combinator	56
15.6 Strict sum preserves flatness	57
16 The Strict Function Type	57

17	Map functions for various types	58
17.1	Map operator for continuous function space	58
17.2	Map operator for product type	59
17.3	Map function for lifted cpo	59
17.4	Map function for strict products	60
17.5	Map function for strict sums	61
17.6	Map operator for strict function space	62
18	The cpo of cartesian products	62
18.1	Continuous case function for unit type	63
18.2	Continuous version of split function	63
18.3	Convert all lemmas to the continuous versions	63
19	Profinite and bifinite cpos	63
19.1	Chains of finite deflations	63
19.2	Omega-profinite and bifinite domains	64
19.3	Building approx chains	64
19.4	Class instance proofs	65
20	Defining algebraic domains by ideal completion	67
20.1	Ideals over a preorder	67
20.2	Lemmas about least upper bounds	68
20.3	Locale for ideal completion	68
20.3.1	Principal ideals approximate all elements	69
20.4	Defining functions in terms of basis elements	70
21	A universal bifinite domain	71
21.1	Basis for universal domain	71
21.1.1	Basis datatype	71
21.1.2	Basis ordering	72
21.1.3	Generic take function	72
21.2	Defining the universal domain by ideal completion	73
21.3	Compact bases of domains	74
21.4	Universality of <i>udom</i>	75
21.4.1	Choosing a maximal element from a finite set	75
21.4.2	Compact basis take function	76
21.4.3	Rank of basis elements	77
21.4.4	Sequencing basis elements	78
21.4.5	Embedding and projection on basis elements	78
21.4.6	EP-pair from any bifinite domain into <i>udom</i>	81
21.5	Chain of approx functions for type <i>udom</i>	81

22 Algebraic deflations	82
22.1 Type constructor for finite deflations	82
22.2 Defining algebraic deflations by ideal completion	83
22.3 Applying algebraic deflations	84
22.4 Deflation combinators	85
23 Representable domains	86
23.1 Class of representable domains	86
23.2 Domains are bifinite	87
23.3 Universal domain ep-pairs	87
23.4 Type combinators	88
23.5 Class instance proofs	89
23.5.1 Universal domain	89
23.5.2 Lifted cpo	90
23.5.3 Strict function space	90
23.5.4 Continuous function space	91
23.5.5 Strict product	92
23.5.6 Cartesian product	92
23.5.7 Unit type	93
23.5.8 Discrete cpo	94
23.5.9 Strict sum	94
23.5.10 Lifted HOL type	95
24 The unit domain	95
25 Fixed point operator and admissibility	97
25.1 Iteration	97
25.2 Least fixed point operator	97
25.3 Fixed point induction	99
25.4 Fixed-points on product types	100
26 Package for defining recursive functions in HOLCF	100
26.1 Pattern-match monad	100
26.1.1 Run operator	101
26.1.2 Monad plus operator	101
26.2 Match functions for built-in types	101
26.3 Mutual recursion	103
26.4 Initializing the fixrec package	104
27 Domain package support	104
27.1 Continuous isomorphisms	104
27.2 Proofs about take functions	106
27.3 Finiteness	106
27.4 Proofs about constructor functions	107

27.5 ML setup	109
28 Domain package	109
28.1 Representations of types	110
28.2 Deflations as sets	110
28.3 Proving a subtype is representable	111
28.4 Isomorphic deflations	111
28.5 Setting up the domain package	113
29 A compact basis for powerdomains	114
29.1 A compact basis for powerdomains	114
29.2 Unit and plus constructors	115
29.3 Fold operator	115
30 Upper powerdomain	116
30.1 Basis preorder	116
30.2 Type definition	117
30.3 Monadic unit and plus	118
30.4 Induction rules	120
30.5 Monadic bind	120
30.6 Map	121
30.7 Upper powerdomain is bifinite	122
30.8 Join	122
31 Lower powerdomain	123
31.1 Basis preorder	123
31.2 Type definition	124
31.3 Monadic unit and plus	125
31.4 Induction rules	127
31.5 Monadic bind	127
31.6 Map	128
31.7 Lower powerdomain is bifinite	129
31.8 Join	129
32 Convex powerdomain	130
32.1 Basis preorder	130
32.2 Type definition	131
32.3 Monadic unit and plus	132
32.4 Induction rules	134
32.5 Monadic bind	134
32.6 Map	135
32.7 Convex powerdomain is bifinite	136
32.8 Join	136
32.9 Conversions to other powerdomains	137

33 Powerdomains	139
33.1 Universal domain embeddings	139
33.2 Deflation combinators	139
33.3 Domain class instances	140
33.4 Isomorphic deflations	142
33.5 Domain package setup for powerdomains	142



1 Partial orders

```
theory Porder
  imports Main
begin
```

```
declare [[typedef-overloaded]]
```

1.1 Type class for partial orders

```
class below =
  fixes below :: 'a ⇒ 'a ⇒ bool
begin
```

```
notation (ASCII)
  below (infix << 50)
```

```
notation
  below (infix ⊑ 50)
```

```
abbreviation not-below :: 'a ⇒ 'a ⇒ bool (infix ≱ 50)
  where not-below x y ≡ ¬ below x y
```

```
notation (ASCII)
  not-below (infix ~<< 50)
```

```
lemma below-eq-trans: a ⊑ b ⇒ b = c ⇒ a ⊑ c
  <proof>
```

```
lemma eq-below-trans: a = b ⇒ b ⊑ c ⇒ a ⊑ c
  <proof>
```

```
end
```

```
class po = below +
  assumes below-refl [iff]: x ⊑ x
  assumes below-trans: x ⊑ y ⇒ y ⊑ z ⇒ x ⊑ z
  assumes below-antisym: x ⊑ y ⇒ y ⊑ x ⇒ x = y
begin
```

```
lemma eq-imp-below: x = y ⇒ x ⊑ y
  <proof>
```

```
lemma box-below: a ⊑ b ⇒ c ⊑ a ⇒ b ⊑ d ⇒ c ⊑ d
  <proof>
```

```
lemma po-eq-conv: x = y ↔ x ⊑ y ∧ y ⊑ x
  <proof>
```

```
lemma rev-below-trans: y ⊑ z ⇒ x ⊑ y ⇒ x ⊑ z
```

<proof>

lemma *not-below2not-eq*: $x \not\sqsubseteq y \implies x \neq y$
<proof>

end

lemmas *HOLCF-trans-rules* [*trans*] =
below-trans
below-antisym
below-eq-trans
eq-below-trans

context *po*
begin

1.2 Upper bounds

definition *is-ub* :: 'a set \Rightarrow 'a \Rightarrow bool (**infix** <| 55)
where $S <| x \longleftrightarrow (\forall y \in S. y \sqsubseteq x)$

lemma *is-ubI*: $(\bigwedge x. x \in S \implies x \sqsubseteq u) \implies S <| u$
<proof>

lemma *is-ubD*: $\llbracket S <| u; x \in S \rrbracket \implies x \sqsubseteq u$
<proof>

lemma *ub-imageI*: $(\bigwedge x. x \in S \implies f x \sqsubseteq u) \implies (\lambda x. f x) ' S <| u$
<proof>

lemma *ub-imageD*: $\llbracket f ' S <| u; x \in S \rrbracket \implies f x \sqsubseteq u$
<proof>

lemma *ub-rangeI*: $(\bigwedge i. S i \sqsubseteq x) \implies \text{range } S <| x$
<proof>

lemma *ub-rangeD*: $\text{range } S <| x \implies S i \sqsubseteq x$
<proof>

lemma *is-ub-empty* [*simp*]: $\{\} <| u$
<proof>

lemma *is-ub-insert* [*simp*]: $(\text{insert } x A) <| y = (x \sqsubseteq y \wedge A <| y)$
<proof>

lemma *is-ub-upward*: $\llbracket S <| x; x \sqsubseteq y \rrbracket \implies S <| y$
<proof>

1.3 Least upper bounds

definition *is-lub* :: 'a set \Rightarrow 'a \Rightarrow bool (**infix** <<| 55)
where $S <<| x \longleftrightarrow S <| x \wedge (\forall u. S <| u \longrightarrow x \sqsubseteq u)$

definition *lub* :: 'a set \Rightarrow 'a
where $lub\ S = (THE\ x.\ S <<| x)$

end

syntax (*ASCII*)

-*BLub* :: [pttrn, 'a set, 'b] \Rightarrow 'b ((*3LUB* -:/ -) [0,0, 10] 10)

syntax

-*BLub* :: [pttrn, 'a set, 'b] \Rightarrow 'b ((*3* \sqcup - \in -) [0,0, 10] 10)

translations

LUB $x:A. t \Rightarrow CONST\ lub\ ((\lambda x. t)\ 'A)$

context *po*

begin

abbreviation *Lub* (**binder** \sqcup 10)

where $\sqcup n. t\ n \equiv lub\ (range\ t)$

notation (*ASCII*)

Lub (**binder** *LUB* 10)

access to some definition as inference rule

lemma *is-lubD1*: $S <<| x \Longrightarrow S <| x$
<proof>

lemma *is-lubD2*: $\llbracket S <<| x; S <| u \rrbracket \Longrightarrow x \sqsubseteq u$
<proof>

lemma *is-lubI*: $\llbracket S <| x; \bigwedge u. S <| u \rrbracket \Longrightarrow x \sqsubseteq u \rrbracket \Longrightarrow S <<| x$
<proof>

lemma *is-lub-below-iff*: $S <<| x \Longrightarrow x \sqsubseteq u \longleftrightarrow S <| u$
<proof>

lubs are unique

lemma *is-lub-unique*: $S <<| x \Longrightarrow S <<| y \Longrightarrow x = y$
<proof>

technical lemmas about *lub* and (<<|)

lemma *is-lub-lub*: $M <<| x \Longrightarrow M <<| lub\ M$
<proof>

lemma *lub-eqI*: $M \ll\mid l \implies \text{lub } M = l$
 ⟨proof⟩

lemma *is-lub-singleton* [*simp*]: $\{x\} \ll\mid x$
 ⟨proof⟩

lemma *lub-singleton* [*simp*]: $\text{lub } \{x\} = x$
 ⟨proof⟩

lemma *is-lub-bin*: $x \sqsubseteq y \implies \{x, y\} \ll\mid y$
 ⟨proof⟩

lemma *lub-bin*: $x \sqsubseteq y \implies \text{lub } \{x, y\} = y$
 ⟨proof⟩

lemma *is-lub-maximal*: $S \ll\mid x \implies x \in S \implies S \ll\mid x$
 ⟨proof⟩

lemma *lub-maximal*: $S \ll\mid x \implies x \in S \implies \text{lub } S = x$
 ⟨proof⟩

1.4 Countable chains

definition *chain* :: $(\text{nat} \Rightarrow 'a) \Rightarrow \text{bool}$

where — Here we use countable chains and I prefer to code them as functions!

chain $Y = (\forall i. Y\ i \sqsubseteq Y\ (\text{Suc } i))$

lemma *chainI*: $(\bigwedge i. Y\ i \sqsubseteq Y\ (\text{Suc } i)) \implies \text{chain } Y$
 ⟨proof⟩

lemma *chainE*: $\text{chain } Y \implies Y\ i \sqsubseteq Y\ (\text{Suc } i)$
 ⟨proof⟩

chains are monotone functions

lemma *chain-mono-less*: $\text{chain } Y \implies i < j \implies Y\ i \sqsubseteq Y\ j$
 ⟨proof⟩

lemma *chain-mono*: $\text{chain } Y \implies i \leq j \implies Y\ i \sqsubseteq Y\ j$
 ⟨proof⟩

lemma *chain-shift*: $\text{chain } Y \implies \text{chain } (\lambda i. Y\ (i + j))$
 ⟨proof⟩

technical lemmas about (least) upper bounds of chains

lemma *is-lub-rangeD1*: $\text{range } S \ll\mid x \implies S\ i \sqsubseteq x$
 ⟨proof⟩

lemma *is-ub-range-shift*: $\text{chain } S \implies \text{range } (\lambda i. S\ (i + j)) \ll\mid x = \text{range } S \ll\mid x$
 ⟨proof⟩

lemma *is-lub-range-shift*: $\text{chain } S \implies \text{range } (\lambda i. S (i + j)) \ll x = \text{range } S \ll x$
 ⟨proof⟩

the lub of a constant chain is the constant

lemma *chain-const* [simp]: $\text{chain } (\lambda i. c)$
 ⟨proof⟩

lemma *is-lub-const*: $\text{range } (\lambda x. c) \ll c$
 ⟨proof⟩

lemma *lub-const* [simp]: $(\bigsqcup i. c) = c$
 ⟨proof⟩

1.5 Finite chains

definition *max-in-chain* :: $\text{nat} \Rightarrow (\text{nat} \Rightarrow 'a) \Rightarrow \text{bool}$
 where — finite chains, needed for monotony of continuous functions
 $\text{max-in-chain } i C \iff (\forall j. i \leq j \longrightarrow C i = C j)$

definition *finite-chain* :: $(\text{nat} \Rightarrow 'a) \Rightarrow \text{bool}$
 where $\text{finite-chain } C = (\text{chain } C \wedge (\exists i. \text{max-in-chain } i C))$

results about finite chains

lemma *max-in-chainI*: $(\bigwedge j. i \leq j \implies Y i = Y j) \implies \text{max-in-chain } i Y$
 ⟨proof⟩

lemma *max-in-chainD*: $\text{max-in-chain } i Y \implies i \leq j \implies Y i = Y j$
 ⟨proof⟩

lemma *finite-chainI*: $\text{chain } C \implies \text{max-in-chain } i C \implies \text{finite-chain } C$
 ⟨proof⟩

lemma *finite-chainE*: $\llbracket \text{finite-chain } C; \bigwedge i. \llbracket \text{chain } C; \text{max-in-chain } i C \rrbracket \implies R \rrbracket$
 ⟨proof⟩

lemma *lub-finch1*: $\text{chain } C \implies \text{max-in-chain } i C \implies \text{range } C \ll C i$
 ⟨proof⟩

lemma *lub-finch2*: $\text{finite-chain } C \implies \text{range } C \ll C$ (LEAST $i. \text{max-in-chain } i C$)
 ⟨proof⟩

lemma *finch-imp-finite-range*: $\text{finite-chain } Y \implies \text{finite } (\text{range } Y)$
 ⟨proof⟩

lemma *finite-range-has-max*:

```

fixes f :: nat => 'a
  and r :: 'a => 'a => bool
assumes mono:  $\bigwedge i j. i \leq j \implies r (f i) (f j)$ 
assumes finite-range: finite (range f)
shows  $\exists k. \forall i. r (f i) (f k)$ 
<proof>

```

```

lemma finite-range-imp-finch: chain Y  $\implies$  finite (range Y)  $\implies$  finite-chain Y
<proof>

```

```

lemma bin-chain:  $x \sqsubseteq y \implies$  chain ( $\lambda i. \text{if } i=0 \text{ then } x \text{ else } y$ )
<proof>

```

```

lemma bin-chainmax:  $x \sqsubseteq y \implies$  max-in-chain (Suc 0) ( $\lambda i. \text{if } i=0 \text{ then } x \text{ else } y$ )
<proof>

```

```

lemma is-lub-bin-chain:  $x \sqsubseteq y \implies$  range ( $\lambda i::nat. \text{if } i=0 \text{ then } x \text{ else } y$ )  $\lll y$ 
<proof>

```

the maximal element in a chain is its lub

```

lemma lub-chain-maxelem:  $Y i = c \implies \forall i. Y i \sqsubseteq c \implies$  lub (range Y) = c
<proof>

```

end

end

2 Classes cpo and pcpo

```

theory Pcpo
  imports Porder
begin

```

2.1 Complete partial orders

The class cpo of chain complete partial orders

```

class cpo = po +
  assumes cpo: chain S  $\implies \exists x. \text{range } S \lll x$ 
begin

```

in cpo's everthing equal to THE lub has lub properties for every chain

```

lemma cpo-lubI: chain S  $\implies$  range S  $\lll$  ( $\bigsqcup i. S i$ )
<proof>

```

```

lemma thelubE:  $\llbracket \text{chain } S; (\bigsqcup i. S i) = l \rrbracket \implies$  range S  $\lll l$ 
<proof>

```

Properties of the lub

lemma *is-ub-thelub*: $\text{chain } S \implies S x \sqsubseteq (\bigsqcup i. S i)$
 ⟨proof⟩

lemma *is-lub-thelub*: $\llbracket \text{chain } S; \text{range } S <| x \rrbracket \implies (\bigsqcup i. S i) \sqsubseteq x$
 ⟨proof⟩

lemma *lub-below-iff*: $\text{chain } S \implies (\bigsqcup i. S i) \sqsubseteq x \iff (\forall i. S i \sqsubseteq x)$
 ⟨proof⟩

lemma *lub-below*: $\llbracket \text{chain } S; \bigwedge i. S i \sqsubseteq x \rrbracket \implies (\bigsqcup i. S i) \sqsubseteq x$
 ⟨proof⟩

lemma *below-lub*: $\llbracket \text{chain } S; x \sqsubseteq S i \rrbracket \implies x \sqsubseteq (\bigsqcup i. S i)$
 ⟨proof⟩

lemma *lub-range-mono*: $\llbracket \text{range } X \subseteq \text{range } Y; \text{chain } Y; \text{chain } X \rrbracket \implies (\bigsqcup i. X i) \sqsubseteq (\bigsqcup i. Y i)$
 ⟨proof⟩

lemma *lub-range-shift*: $\text{chain } Y \implies (\bigsqcup i. Y (i + j)) = (\bigsqcup i. Y i)$
 ⟨proof⟩

lemma *maxinch-is-thelub*: $\text{chain } Y \implies \text{max-in-chain } i Y = ((\bigsqcup i. Y i) = Y i)$
 ⟨proof⟩

the \sqsubseteq relation between two chains is preserved by their lubs

lemma *lub-mono*: $\llbracket \text{chain } X; \text{chain } Y; \bigwedge i. X i \sqsubseteq Y i \rrbracket \implies (\bigsqcup i. X i) \sqsubseteq (\bigsqcup i. Y i)$
 ⟨proof⟩

the $=$ relation between two chains is preserved by their lubs

lemma *lub-eq*: $(\bigwedge i. X i = Y i) \implies (\bigsqcup i. X i) = (\bigsqcup i. Y i)$
 ⟨proof⟩

lemma *ch2ch-lub*:

assumes 1: $\bigwedge j. \text{chain } (\lambda i. Y i j)$

assumes 2: $\bigwedge i. \text{chain } (\lambda j. Y i j)$

shows $\text{chain } (\lambda i. \bigsqcup j. Y i j)$

⟨proof⟩

lemma *diag-lub*:

assumes 1: $\bigwedge j. \text{chain } (\lambda i. Y i j)$

assumes 2: $\bigwedge i. \text{chain } (\lambda j. Y i j)$

shows $(\bigsqcup i. \bigsqcup j. Y i j) = (\bigsqcup i. Y i i)$

⟨proof⟩

lemma *ex-lub*:

assumes 1: $\bigwedge j. \text{chain } (\lambda i. Y i j)$

assumes 2: $\bigwedge i. \text{chain } (\lambda j. Y i j)$

shows $(\bigsqcup i. \bigsqcup j. Y i j) = (\bigsqcup j. \bigsqcup i. Y i j)$

<proof>

end

2.2 Pointed cpos

The class pcpo of pointed cpos

class *pcpo* = *cpo* +
assumes *least*: $\exists x. \forall y. x \sqsubseteq y$
begin

definition *bottom* :: 'a (\perp)
where *bottom* = (*THE* $x. \forall y. x \sqsubseteq y$)

lemma *minimal* [*iff*]: $\perp \sqsubseteq x$
<proof>

end

Old "UU" syntax:

syntax *UU* :: *logic*
translations *UU* \rightarrow *CONST bottom*

Simproc to rewrite $\perp = x$ to $x = \perp$.

<ML>

useful lemmas about \perp

lemma *below-bottom-iff* [*simp*]: $x \sqsubseteq \perp \longleftrightarrow x = \perp$
<proof>

lemma *eq-bottom-iff*: $x = \perp \longleftrightarrow x \sqsubseteq \perp$
<proof>

lemma *bottomI*: $x \sqsubseteq \perp \Longrightarrow x = \perp$
<proof>

lemma *lub-eq-bottom-iff*: $\text{chain } Y \Longrightarrow (\bigsqcup i. Y i) = \perp \longleftrightarrow (\forall i. Y i = \perp)$
<proof>

2.3 Chain-finite and flat cpos

further useful classes for HOLCF domains

class *chfin* = *po* +
assumes *chfin*: $\text{chain } Y \Longrightarrow \exists n. \text{max-in-chain } n Y$
begin

subclass *cpo*
<proof>

lemma *chfin2finch*: $\text{chain } Y \implies \text{finite-chain } Y$
 ⟨*proof*⟩

end

class *flat* = *pcpo* +
assumes *ax-flat*: $x \sqsubseteq y \implies x = \perp \vee x = y$
begin

subclass *chfin*
 ⟨*proof*⟩

lemma *flat-below-iff*: $x \sqsubseteq y \longleftrightarrow x = \perp \vee x = y$
 ⟨*proof*⟩

lemma *flat-eq*: $a \neq \perp \implies a \sqsubseteq b = (a = b)$
 ⟨*proof*⟩

end

2.4 Discrete cpos

class *discrete-cpo* = *below* +
assumes *discrete-cpo [simp]*: $x \sqsubseteq y \longleftrightarrow x = y$
begin

subclass *po*
 ⟨*proof*⟩

In a discrete cpo, every chain is constant

lemma *discrete-chain-const*:
assumes *S*: $\text{chain } S$
shows $\exists x. S = (\lambda i. x)$
 ⟨*proof*⟩

subclass *chfin*
 ⟨*proof*⟩

end

end

3 Continuity and monotonicity

theory *Cont*
imports *Pcpo*
begin

Now we change the default class! From now on all untyped type variables are of default class `po`

`default-sort po`

3.1 Definitions

definition *monofun* :: ('a ⇒ 'b) ⇒ bool — monotonicity
where *monofun* f ↔ (∀ x y. x ⊆ y ⇒ f x ⊆ f y)

definition *cont* :: ('a::cpo ⇒ 'b::cpo) ⇒ bool
where *cont* f = (∀ Y. chain Y ⇒ range (λi. f (Y i)) <<| f (⊔ i. Y i))

lemma *contI*: (⋀ Y. chain Y ⇒ range (λi. f (Y i)) <<| f (⊔ i. Y i)) ⇒ *cont* f
 ⟨proof⟩

lemma *contE*: *cont* f ⇒ chain Y ⇒ range (λi. f (Y i)) <<| f (⊔ i. Y i)
 ⟨proof⟩

lemma *monofunI*: (⋀ x y. x ⊆ y ⇒ f x ⊆ f y) ⇒ *monofun* f
 ⟨proof⟩

lemma *monofunE*: *monofun* f ⇒ x ⊆ y ⇒ f x ⊆ f y
 ⟨proof⟩

3.2 Equivalence of alternate definition

monotone functions map chains to chains

lemma *ch2ch-monofun*: *monofun* f ⇒ chain Y ⇒ chain (λi. f (Y i))
 ⟨proof⟩

monotone functions map upper bound to upper bounds

lemma *ub2ub-monofun*: *monofun* f ⇒ range Y <| u ⇒ range (λi. f (Y i)) <| f u
 ⟨proof⟩

a lemma about binary chains

lemma *binchain-cont*: *cont* f ⇒ x ⊆ y ⇒ range (λi::nat. f (if i = 0 then x else y)) <<| f y
 ⟨proof⟩

continuity implies monotonicity

lemma *cont2mono*: *cont* f ⇒ *monofun* f
 ⟨proof⟩

lemmas *cont2monofunE* = *cont2mono* [THEN *monofunE*]

lemmas *ch2ch-cont* = *cont2mono* [THEN *ch2ch-monofun*]

continuity implies preservation of lubs

lemma *cont2contlubE*: $cont\ f \implies chain\ Y \implies f\ (\sqcup i.\ Y\ i) = (\sqcup i.\ f\ (Y\ i))$
 ⟨*proof*⟩

lemma *contI2*:

fixes $f :: 'a::cpo \Rightarrow 'b::cpo$
assumes *mono*: *monofun* f
assumes *below*: $\bigwedge Y. \llbracket chain\ Y; chain\ (\lambda i.\ f\ (Y\ i)) \rrbracket \implies f\ (\sqcup i.\ Y\ i) \sqsubseteq (\sqcup i.\ f\ (Y\ i))$
shows $cont\ f$
 ⟨*proof*⟩

3.3 Collection of continuity rules

named-theorems *cont2cont* *continuity* *intro* *rule*

3.4 Continuity of basic functions

The identity function is continuous

lemma *cont-id* [*simp*, *cont2cont*]: $cont\ (\lambda x.\ x)$
 ⟨*proof*⟩

constant functions are continuous

lemma *cont-const* [*simp*, *cont2cont*]: $cont\ (\lambda x.\ c)$
 ⟨*proof*⟩

application of functions is continuous

lemma *cont-apply*:

fixes $f :: 'a::cpo \Rightarrow 'b::cpo \Rightarrow 'c::cpo$ **and** $t :: 'a \Rightarrow 'b$
assumes *1*: $cont\ (\lambda x.\ t\ x)$
assumes *2*: $\bigwedge x.\ cont\ (\lambda y.\ f\ x\ y)$
assumes *3*: $\bigwedge y.\ cont\ (\lambda x.\ f\ x\ y)$
shows $cont\ (\lambda x.\ (f\ x)\ (t\ x))$
 ⟨*proof*⟩

lemma *cont-compose*: $cont\ c \implies cont\ (\lambda x.\ f\ x) \implies cont\ (\lambda x.\ c\ (f\ x))$
 ⟨*proof*⟩

Least upper bounds preserve continuity

lemma *cont2cont-lub* [*simp*]:

assumes *chain*: $\bigwedge x.\ chain\ (\lambda i.\ F\ i\ x)$
and *cont*: $\bigwedge i.\ cont\ (\lambda x.\ F\ i\ x)$
shows $cont\ (\lambda x.\ \sqcup i.\ F\ i\ x)$
 ⟨*proof*⟩

if-then-else is continuous

lemma *cont-if* [*simp*, *cont2cont*]: $cont\ f \implies cont\ g \implies cont\ (\lambda x.\ if\ b\ then\ f\ x\ else\ g\ x)$
 ⟨*proof*⟩

3.5 Finite chains and flat pcpos

Monotone functions map finite chains to finite chains.

lemma *monofun-finch2finch*: $\text{monofun } f \implies \text{finite-chain } Y \implies \text{finite-chain } (\lambda n. f (Y n))$
 ⟨proof⟩

The same holds for continuous functions.

lemma *cont-finch2finch*: $\text{cont } f \implies \text{finite-chain } Y \implies \text{finite-chain } (\lambda n. f (Y n))$
 ⟨proof⟩

All monotone functions with chain-finite domain are continuous.

lemma *chfindom-monofun2cont*: $\text{monofun } f \implies \text{cont } f$
 for $f :: 'a::\text{chfin} \Rightarrow 'b::\text{cpo}$
 ⟨proof⟩

All strict functions with flat domain are continuous.

lemma *flatdom-strict2mono*: $f \perp = \perp \implies \text{monofun } f$
 for $f :: 'a::\text{flat} \Rightarrow 'b::\text{pcpo}$
 ⟨proof⟩

lemma *flatdom-strict2cont*: $f \perp = \perp \implies \text{cont } f$
 for $f :: 'a::\text{flat} \Rightarrow 'b::\text{pcpo}$
 ⟨proof⟩

All functions with discrete domain are continuous.

lemma *cont-discrete-cpo* [*simp*, *cont2cont*]: $\text{cont } f$
 for $f :: 'a::\text{discrete-cpo} \Rightarrow 'b::\text{cpo}$
 ⟨proof⟩

end

4 Admissibility and compactness

theory *Adm*
 imports *Cont*
 begin

default-sort *cpo*

4.1 Definitions

definition *adm* :: $('a::\text{cpo} \Rightarrow \text{bool}) \Rightarrow \text{bool}$
 where $\text{adm } P \iff (\forall Y. \text{chain } Y \longrightarrow (\forall i. P (Y i)) \longrightarrow P (\bigsqcup i. Y i))$

lemma *admI*: $(\bigwedge Y. \llbracket \text{chain } Y; \forall i. P (Y i) \rrbracket \implies P (\bigsqcup i. Y i)) \implies \text{adm } P$
 ⟨proof⟩

lemma *admD*: $adm\ P \implies chain\ Y \implies (\bigwedge i. P\ (Y\ i)) \implies P\ (\bigsqcup i. Y\ i)$
 ⟨proof⟩

lemma *admD2*: $adm\ (\lambda x. \neg P\ x) \implies chain\ Y \implies P\ (\bigsqcup i. Y\ i) \implies \exists i. P\ (Y\ i)$
 ⟨proof⟩

lemma *triv-admI*: $\forall x. P\ x \implies adm\ P$
 ⟨proof⟩

4.2 Admissibility on chain-finite types

For chain-finite (easy) types every formula is admissible.

lemma *adm-chfin* [*simp*]: $adm\ P$
 for $P :: 'a::chfin \Rightarrow bool$
 ⟨proof⟩

4.3 Admissibility of special formulae and propagation

lemma *adm-const* [*simp*]: $adm\ (\lambda x. t)$
 ⟨proof⟩

lemma *adm-conj* [*simp*]: $adm\ (\lambda x. P\ x) \implies adm\ (\lambda x. Q\ x) \implies adm\ (\lambda x. P\ x \wedge Q\ x)$
 ⟨proof⟩

lemma *adm-all* [*simp*]: $(\bigwedge y. adm\ (\lambda x. P\ x\ y)) \implies adm\ (\lambda x. \forall y. P\ x\ y)$
 ⟨proof⟩

lemma *adm-ball* [*simp*]: $(\bigwedge y. y \in A \implies adm\ (\lambda x. P\ x\ y)) \implies adm\ (\lambda x. \forall y \in A. P\ x\ y)$
 ⟨proof⟩

Admissibility for disjunction is hard to prove. It requires 2 lemmas.

lemma *adm-disj-lemma1*:
 assumes *adm*: $adm\ P$
 assumes *chain*: $chain\ Y$
 assumes *P*: $\forall i. \exists j \geq i. P\ (Y\ j)$
 shows $P\ (\bigsqcup i. Y\ i)$
 ⟨proof⟩

lemma *adm-disj-lemma2*: $\forall n::nat. P\ n \vee Q\ n \implies (\forall i. \exists j \geq i. P\ j) \vee (\forall i. \exists j \geq i. Q\ j)$
 ⟨proof⟩

lemma *adm-disj* [*simp*]: $adm\ (\lambda x. P\ x) \implies adm\ (\lambda x. Q\ x) \implies adm\ (\lambda x. P\ x \vee Q\ x)$
 ⟨proof⟩

lemma *adm-imp* [*simp*]: $adm (\lambda x. \neg P x) \Longrightarrow adm (\lambda x. Q x) \Longrightarrow adm (\lambda x. P x \longrightarrow Q x)$
 ⟨*proof*⟩

lemma *adm-iff* [*simp*]: $adm (\lambda x. P x \longrightarrow Q x) \Longrightarrow adm (\lambda x. Q x \longrightarrow P x) \Longrightarrow adm (\lambda x. P x \longleftrightarrow Q x)$
 ⟨*proof*⟩

admissibility and continuity

lemma *adm-below* [*simp*]: $cont (\lambda x. u x) \Longrightarrow cont (\lambda x. v x) \Longrightarrow adm (\lambda x. u x \sqsubseteq v x)$
 ⟨*proof*⟩

lemma *adm-eq* [*simp*]: $cont (\lambda x. u x) \Longrightarrow cont (\lambda x. v x) \Longrightarrow adm (\lambda x. u x = v x)$
 ⟨*proof*⟩

lemma *adm-subst*: $cont (\lambda x. t x) \Longrightarrow adm P \Longrightarrow adm (\lambda x. P (t x))$
 ⟨*proof*⟩

lemma *adm-not-below* [*simp*]: $cont (\lambda x. t x) \Longrightarrow adm (\lambda x. t x \not\sqsubseteq u)$
 ⟨*proof*⟩

4.4 Compactness

definition *compact* :: $'a::cpo \Rightarrow bool$
 where *compact* $k = adm (\lambda x. k \not\sqsubseteq x)$

lemma *compactI*: $adm (\lambda x. k \not\sqsubseteq x) \Longrightarrow compact k$
 ⟨*proof*⟩

lemma *compactD*: $compact k \Longrightarrow adm (\lambda x. k \not\sqsubseteq x)$
 ⟨*proof*⟩

lemma *compactI2*: $(\bigwedge Y. \llbracket chain Y; x \sqsubseteq (\bigsqcup i. Y i) \rrbracket \Longrightarrow \exists i. x \sqsubseteq Y i) \Longrightarrow compact x$
 ⟨*proof*⟩

lemma *compactD2*: $compact x \Longrightarrow chain Y \Longrightarrow x \sqsubseteq (\bigsqcup i. Y i) \Longrightarrow \exists i. x \sqsubseteq Y i$
 ⟨*proof*⟩

lemma *compact-below-lub-iff*: $compact x \Longrightarrow chain Y \Longrightarrow x \sqsubseteq (\bigsqcup i. Y i) \longleftrightarrow (\exists i. x \sqsubseteq Y i)$
 ⟨*proof*⟩

lemma *compact-chfin* [*simp*]: *compact* x
 for $x :: 'a::chfin$
 ⟨*proof*⟩

lemma *compact-imp-max-in-chain*: $chain Y \Longrightarrow compact (\bigsqcup i. Y i) \Longrightarrow \exists i. max-in-chain$

i Y
 ⟨proof⟩

admissibility and compactness

lemma *adm-compact-not-below* [*simp*]:
 $compact\ k \implies cont\ (\lambda x. t\ x) \implies adm\ (\lambda x. k \not\sqsubseteq t\ x)$
 ⟨proof⟩

lemma *adm-neq-compact* [*simp*]: $compact\ k \implies cont\ (\lambda x. t\ x) \implies adm\ (\lambda x. t\ x \neq k)$
 ⟨proof⟩

lemma *adm-compact-neq* [*simp*]: $compact\ k \implies cont\ (\lambda x. t\ x) \implies adm\ (\lambda x. k \neq t\ x)$
 ⟨proof⟩

lemma *compact-bottom* [*simp, intro*]: $compact\ \perp$
 ⟨proof⟩

Any upward-closed predicate is admissible.

lemma *adm-upward*:
assumes $P: \bigwedge x\ y. [P\ x; x \sqsubseteq y] \implies P\ y$
shows $adm\ P$
 ⟨proof⟩

lemmas *adm-lemmas* =
adm-const adm-conj adm-all adm-ball adm-disj adm-imp adm-iff
adm-below adm-eq adm-not-below
adm-compact-not-below adm-compact-neq adm-neq-compact

end

5 Subtypes of pcpo

theory *Cpodef*
imports *Adm*
keywords *pcpodef cpodef :: thy-goal-defn*
begin

5.1 Proving a subtype is a partial order

A subtype of a partial order is itself a partial order, if the ordering is defined in the standard way.

⟨ML⟩

theorem *typedef-po*:
fixes $Abs :: 'a::po \Rightarrow 'b::type$
assumes $type: type-definition\ Rep\ Abs\ A$

and below: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
shows $\text{OFCLASS}('b, \text{po-class})$
 $\langle \text{proof} \rangle$

$\langle \text{ML} \rangle$

5.2 Proving a subtype is finite

lemma *typedef-finite-UNIV*:
fixes $\text{Abs} :: 'a::\text{type} \Rightarrow 'b::\text{type}$
assumes $\text{type: type-definition Rep Abs A}$
shows $\text{finite } A \Longrightarrow \text{finite } (\text{UNIV} :: 'b \text{ set})$
 $\langle \text{proof} \rangle$

5.3 Proving a subtype is chain-finite

lemma *ch2ch-Rep*:
assumes $\text{below: } (\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
shows $\text{chain } S \Longrightarrow \text{chain } (\lambda i. \text{Rep } (S i))$
 $\langle \text{proof} \rangle$

theorem *typedef-chfin*:
fixes $\text{Abs} :: 'a::\text{chfin} \Rightarrow 'b::\text{po}$
assumes $\text{type: type-definition Rep Abs A}$
and below: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
shows $\text{OFCLASS}('b, \text{chfin-class})$
 $\langle \text{proof} \rangle$

5.4 Proving a subtype is complete

A subtype of a cpo is itself a cpo if the ordering is defined in the standard way, and the defining subset is closed with respect to limits of chains. A set is closed if and only if membership in the set is an admissible predicate.

lemma *typedef-is-lubI*:
assumes $\text{below: } (\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
shows $\text{range } (\lambda i. \text{Rep } (S i)) \ll\ll \text{Rep } x \Longrightarrow \text{range } S \ll\ll x$
 $\langle \text{proof} \rangle$

lemma *Abs-inverse-lub-Rep*:
fixes $\text{Abs} :: 'a::\text{cpo} \Rightarrow 'b::\text{po}$
assumes $\text{type: type-definition Rep Abs A}$
and below: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
and adm: $\text{adm } (\lambda x. x \in A)$
shows $\text{chain } S \Longrightarrow \text{Rep } (\text{Abs } (\bigsqcup i. \text{Rep } (S i))) = (\bigsqcup i. \text{Rep } (S i))$
 $\langle \text{proof} \rangle$

theorem *typedef-is-lub*:
fixes $\text{Abs} :: 'a::\text{cpo} \Rightarrow 'b::\text{po}$
assumes $\text{type: type-definition Rep Abs A}$

and below: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
and adm: $\text{adm } (\lambda x. x \in A)$
assumes S : *chain* S
shows $\text{range } S \ll \text{Abs } (\bigsqcup i. \text{Rep } (S i))$
 ⟨*proof*⟩

lemmas *typedef-lub* = *typedef-is-lub* [THEN *lub-eqI*]

theorem *typedef-cpo*:
fixes $\text{Abs} :: 'a::\text{cpo} \Rightarrow 'b::\text{po}$
assumes type : *type-definition* $\text{Rep } \text{Abs } A$
and below: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
and adm: $\text{adm } (\lambda x. x \in A)$
shows $\text{OFCLASS}('b, \text{cpo-class})$
 ⟨*proof*⟩

5.4.1 Continuity of *Rep* and *Abs*

For any sub-cpo, the *Rep* function is continuous.

theorem *typedef-cont-Rep*:
fixes $\text{Abs} :: 'a::\text{cpo} \Rightarrow 'b::\text{cpo}$
assumes type : *type-definition* $\text{Rep } \text{Abs } A$
and below: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
and adm: $\text{adm } (\lambda x. x \in A)$
shows $\text{cont } (\lambda x. f x) \Longrightarrow \text{cont } (\lambda x. \text{Rep } (f x))$
 ⟨*proof*⟩

For a sub-cpo, we can make the *Abs* function continuous only if we restrict its domain to the defining subset by composing it with another continuous function.

theorem *typedef-cont-Abs*:
fixes $\text{Abs} :: 'a::\text{cpo} \Rightarrow 'b::\text{cpo}$
fixes $f :: 'c::\text{cpo} \Rightarrow 'a::\text{cpo}$
assumes type : *type-definition* $\text{Rep } \text{Abs } A$
and below: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
and adm: $\text{adm } (\lambda x. x \in A)$
and f-in-A: $\bigwedge x. f x \in A$
shows $\text{cont } f \Longrightarrow \text{cont } (\lambda x. \text{Abs } (f x))$
 ⟨*proof*⟩

5.5 Proving subtype elements are compact

theorem *typedef-compact*:
fixes $\text{Abs} :: 'a::\text{cpo} \Rightarrow 'b::\text{cpo}$
assumes type : *type-definition* $\text{Rep } \text{Abs } A$
and below: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
and adm: $\text{adm } (\lambda x. x \in A)$
shows $\text{compact } (\text{Rep } k) \Longrightarrow \text{compact } k$
 ⟨*proof*⟩

5.6 Proving a subtype is pointed

A subtype of a cpo has a least element if and only if the defining subset has a least element.

theorem *typedef-pcpo-generic:*

fixes $Abs :: 'a::cpo \Rightarrow 'b::cpo$
assumes *type: type-definition Rep Abs A*
and below: $(\sqsubseteq) \equiv \lambda x y. Rep\ x \sqsubseteq Rep\ y$
and z-in-A: $z \in A$
and z-least: $\bigwedge x. x \in A \implies z \sqsubseteq x$
shows $OFCLASS('b, pcpo-class)$
<proof>

As a special case, a subtype of a pcpo has a least element if the defining subset contains \perp .

theorem *typedef-pcpo:*

fixes $Abs :: 'a::pcpo \Rightarrow 'b::cpo$
assumes *type: type-definition Rep Abs A*
and below: $(\sqsubseteq) \equiv \lambda x y. Rep\ x \sqsubseteq Rep\ y$
and bottom-in-A: $\perp \in A$
shows $OFCLASS('b, pcpo-class)$
<proof>

5.6.1 Strictness of *Rep* and *Abs*

For a sub-pcpo where \perp is a member of the defining subset, *Rep* and *Abs* are both strict.

theorem *typedef-Abs-strict:*

assumes *type: type-definition Rep Abs A*
and below: $(\sqsubseteq) \equiv \lambda x y. Rep\ x \sqsubseteq Rep\ y$
and bottom-in-A: $\perp \in A$
shows $Abs\ \perp = \perp$
<proof>

theorem *typedef-Rep-strict:*

assumes *type: type-definition Rep Abs A*
and below: $(\sqsubseteq) \equiv \lambda x y. Rep\ x \sqsubseteq Rep\ y$
and bottom-in-A: $\perp \in A$
shows $Rep\ \perp = \perp$
<proof>

theorem *typedef-Abs-bottom-iff:*

assumes *type: type-definition Rep Abs A*
and below: $(\sqsubseteq) \equiv \lambda x y. Rep\ x \sqsubseteq Rep\ y$
and bottom-in-A: $\perp \in A$
shows $x \in A \implies (Abs\ x = \perp) = (x = \perp)$
<proof>

theorem *typedef-Rep-bottom-iff*:
assumes *type*: *type-definition* *Rep* *Abs* *A*
and *below*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
and *bottom-in-A*: $\perp \in A$
shows $(\text{Rep } x = \perp) = (x = \perp)$
 $\langle \text{proof} \rangle$

5.7 Proving a subtype is flat

theorem *typedef-flat*:
fixes *Abs* :: *'a::flat* \Rightarrow *'b::pcpo*
assumes *type*: *type-definition* *Rep* *Abs* *A*
and *below*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
and *bottom-in-A*: $\perp \in A$
shows *OFCLASS*(*'b*, *flat-class*)
 $\langle \text{proof} \rangle$

5.8 HOLCF type definition package

$\langle ML \rangle$

end

6 Class instances for the full function space

theory *Fun-Cpo*
imports *Adm*
begin

6.1 Full function space is a partial order

instantiation *fun* :: (*type*, *below*) *below*
begin

definition *below-fun-def*: $(\sqsubseteq) \equiv (\lambda f g. \forall x. f x \sqsubseteq g x)$

instance $\langle \text{proof} \rangle$
end

instance *fun* :: (*type*, *po*) *po*
 $\langle \text{proof} \rangle$

lemma *fun-below-iff*: $f \sqsubseteq g \longleftrightarrow (\forall x. f x \sqsubseteq g x)$
 $\langle \text{proof} \rangle$

lemma *fun-belowI*: $(\bigwedge x. f x \sqsubseteq g x) \Longrightarrow f \sqsubseteq g$
 $\langle \text{proof} \rangle$

lemma *fun-belowD*: $f \sqsubseteq g \Longrightarrow f x \sqsubseteq g x$

<proof>

6.2 Full function space is chain complete

Properties of chains of functions.

lemma *fun-chain-iff*: $chain\ S \longleftrightarrow (\forall x. chain\ (\lambda i. S\ i\ x))$
<proof>

lemma *ch2ch-fun*: $chain\ S \implies chain\ (\lambda i. S\ i\ x)$
<proof>

lemma *ch2ch-lambda*: $(\bigwedge x. chain\ (\lambda i. S\ i\ x)) \implies chain\ S$
<proof>

Type $'a \Rightarrow 'b$ is chain complete

lemma *is-lub-lambda*: $(\bigwedge x. range\ (\lambda i. Y\ i\ x) \ll\!| f\ x) \implies range\ Y \ll\!| f$
<proof>

lemma *is-lub-fun*: $chain\ S \implies range\ S \ll\!| (\lambda x. \bigsqcup i. S\ i\ x)$
for $S :: nat \Rightarrow 'a::type \Rightarrow 'b::cpo$
<proof>

lemma *lub-fun*: $chain\ S \implies (\bigsqcup i. S\ i) = (\lambda x. \bigsqcup i. S\ i\ x)$
for $S :: nat \Rightarrow 'a::type \Rightarrow 'b::cpo$
<proof>

instance *fun* :: $(type, cpo)\ cpo$
<proof>

instance *fun* :: $(type, discrete-cpo)\ discrete-cpo$
<proof>

6.3 Full function space is pointed

lemma *minimal-fun*: $(\lambda x. \perp) \sqsubseteq f$
<proof>

instance *fun* :: $(type, pcpo)\ pcpo$
<proof>

lemma *inst-fun-pcpo*: $\perp = (\lambda x. \perp)$
<proof>

lemma *app-strict* [*simp*]: $\perp\ x = \perp$
<proof>

lemma *lambda-strict*: $(\lambda x. \perp) = \perp$
<proof>

6.4 Propagation of monotonicity and continuity

The lub of a chain of monotone functions is monotone.

lemma *adm-monofun*: *adm monofun*
 ⟨*proof*⟩

The lub of a chain of continuous functions is continuous.

lemma *adm-cont*: *adm cont*
 ⟨*proof*⟩

Function application preserves monotonicity and continuity.

lemma *mono2mono-fun*: *monofun f* \implies *monofun* $(\lambda x. f x y)$
 ⟨*proof*⟩

lemma *cont2cont-fun*: *cont f* \implies *cont* $(\lambda x. f x y)$
 ⟨*proof*⟩

lemma *cont-fun*: *cont* $(\lambda f. f x)$
 ⟨*proof*⟩

Lambda abstraction preserves monotonicity and continuity. (Note $(\lambda x. \lambda y. f x y) = f$.)

lemma *mono2mono-lambda*: $(\bigwedge y. \text{monofun } (\lambda x. f x y)) \implies \text{monofun } f$
 ⟨*proof*⟩

lemma *cont2cont-lambda* [*simp*]:
assumes *f*: $\bigwedge y. \text{cont } (\lambda x. f x y)$
shows *cont f*
 ⟨*proof*⟩

What D.A.Schmidt calls continuity of abstraction; never used here

lemma *contlub-lambda*: $(\bigwedge x. \text{chain } (\lambda i. S i x)) \implies (\lambda x. \bigsqcup i. S i x) = (\bigsqcup i. (\lambda x. S i x))$
for *S* :: *nat* \Rightarrow '*a*::*type* \Rightarrow '*b*::*cpo*
 ⟨*proof*⟩

end

7 The cpo of cartesian products

```
theory Product-Cpo
  imports Adm
begin

default-sort cpo
```

7.1 Unit type is a pcpo

instantiation *unit* :: *discrete-cpo*
begin

definition *below-unit-def* [*simp*]: $x \sqsubseteq (y::\text{unit}) \longleftrightarrow \text{True}$

instance
 ⟨*proof*⟩

end

instance *unit* :: *pcpo*
 ⟨*proof*⟩

7.2 Product type is a partial order

instantiation *prod* :: (*below*, *below*) *below*
begin

definition *below-prod-def*: $(\sqsubseteq) \equiv \lambda p1\ p2. (fst\ p1 \sqsubseteq fst\ p2 \wedge snd\ p1 \sqsubseteq snd\ p2)$

instance ⟨*proof*⟩

end

instance *prod* :: (*po*, *po*) *po*
 ⟨*proof*⟩

7.3 Monotonicity of *Pair*, *fst*, *snd*

lemma *prod-belowI*: $fst\ p \sqsubseteq fst\ q \implies snd\ p \sqsubseteq snd\ q \implies p \sqsubseteq q$
 ⟨*proof*⟩

lemma *Pair-below-iff* [*simp*]: $(a, b) \sqsubseteq (c, d) \longleftrightarrow a \sqsubseteq c \wedge b \sqsubseteq d$
 ⟨*proof*⟩

Pair (-,-) is monotone in both arguments

lemma *monofun-pair1*: *monofun* ($\lambda x. (x, y)$)
 ⟨*proof*⟩

lemma *monofun-pair2*: *monofun* ($\lambda y. (x, y)$)
 ⟨*proof*⟩

lemma *monofun-pair*: $x1 \sqsubseteq x2 \implies y1 \sqsubseteq y2 \implies (x1, y1) \sqsubseteq (x2, y2)$
 ⟨*proof*⟩

lemma *ch2ch-Pair* [*simp*]: $chain\ X \implies chain\ Y \implies chain\ (\lambda i. (X\ i, Y\ i))$
 ⟨*proof*⟩

fst and *snd* are monotone

lemma *fst-monofun*: $x \sqsubseteq y \implies \text{fst } x \sqsubseteq \text{fst } y$
 ⟨*proof*⟩

lemma *snd-monofun*: $x \sqsubseteq y \implies \text{snd } x \sqsubseteq \text{snd } y$
 ⟨*proof*⟩

lemma *monofun-fst*: *monofun fst*
 ⟨*proof*⟩

lemma *monofun-snd*: *monofun snd*
 ⟨*proof*⟩

lemmas *ch2ch-fst* [*simp*] = *ch2ch-monofun* [*OF monofun-fst*]

lemmas *ch2ch-snd* [*simp*] = *ch2ch-monofun* [*OF monofun-snd*]

lemma *prod-chain-cases*:

assumes *chain*: *chain Y*

obtains *A B*

where *chain A* and *chain B* and $Y = (\lambda i. (A \ i, B \ i))$

⟨*proof*⟩

7.4 Product type is a cpo

lemma *is-lub-Pair*: $\text{range } A \ll\!| \ x \implies \text{range } B \ll\!| \ y \implies \text{range } (\lambda i. (A \ i, B \ i)) \ll\!| \ (x, y)$
 ⟨*proof*⟩

lemma *lub-Pair*: $\text{chain } A \implies \text{chain } B \implies (\bigsqcup i. (A \ i, B \ i)) = (\bigsqcup i. A \ i, \bigsqcup i. B \ i)$
for $A :: \text{nat} \Rightarrow 'a::\text{cpo}$ and $B :: \text{nat} \Rightarrow 'b::\text{cpo}$
 ⟨*proof*⟩

lemma *is-lub-prod*:

fixes $S :: \text{nat} \Rightarrow ('a::\text{cpo} \times 'b::\text{cpo})$

assumes *chain S*

shows $\text{range } S \ll\!| \ (\bigsqcup i. \text{fst } (S \ i), \bigsqcup i. \text{snd } (S \ i))$

⟨*proof*⟩

lemma *lub-prod*: $\text{chain } S \implies (\bigsqcup i. S \ i) = (\bigsqcup i. \text{fst } (S \ i), \bigsqcup i. \text{snd } (S \ i))$
for $S :: \text{nat} \Rightarrow 'a::\text{cpo} \times 'b::\text{cpo}$
 ⟨*proof*⟩

instance *prod* :: $(\text{cpo}, \text{cpo}) \text{ cpo}$
 ⟨*proof*⟩

instance *prod* :: $(\text{discrete-cpo}, \text{discrete-cpo}) \text{ discrete-cpo}$
 ⟨*proof*⟩

7.5 Product type is pointed

lemma *minimal-prod*: $(\perp, \perp) \sqsubseteq p$
 ⟨proof⟩

instance *prod* :: (pcpo, pcpo) pcpo
 ⟨proof⟩

lemma *inst-prod-pcpo*: $\perp = (\perp, \perp)$
 ⟨proof⟩

lemma *Pair-bottom-iff* [*simp*]: $(x, y) = \perp \iff x = \perp \wedge y = \perp$
 ⟨proof⟩

lemma *fst-strict* [*simp*]: $\text{fst } \perp = \perp$
 ⟨proof⟩

lemma *snd-strict* [*simp*]: $\text{snd } \perp = \perp$
 ⟨proof⟩

lemma *Pair-strict* [*simp*]: $(\perp, \perp) = \perp$
 ⟨proof⟩

lemma *split-strict* [*simp*]: $\text{case-prod } f \perp = f \perp \perp$
 ⟨proof⟩

7.6 Continuity of *Pair*, *fst*, *snd*

lemma *cont-pair1*: $\text{cont } (\lambda x. (x, y))$
 ⟨proof⟩

lemma *cont-pair2*: $\text{cont } (\lambda y. (x, y))$
 ⟨proof⟩

lemma *cont-fst*: $\text{cont } \text{fst}$
 ⟨proof⟩

lemma *cont-snd*: $\text{cont } \text{snd}$
 ⟨proof⟩

lemma *cont2cont-Pair* [*simp*, *cont2cont*]:
assumes $f: \text{cont } (\lambda x. f x)$
assumes $g: \text{cont } (\lambda x. g x)$
shows $\text{cont } (\lambda x. (f x, g x))$
 ⟨proof⟩

lemmas *cont2cont-fst* [*simp*, *cont2cont*] = *cont-compose* [*OF cont-fst*]

lemmas *cont2cont-snd* [*simp*, *cont2cont*] = *cont-compose* [*OF cont-snd*]

lemma *cont2cont-case-prod*:
assumes $f1: \bigwedge a b. \text{cont } (\lambda x. f x a b)$
assumes $f2: \bigwedge x b. \text{cont } (\lambda a. f x a b)$
assumes $f3: \bigwedge x a. \text{cont } (\lambda b. f x a b)$
assumes $g: \text{cont } (\lambda x. g x)$
shows $\text{cont } (\lambda x. \text{case } g x \text{ of } (a, b) \Rightarrow f x a b)$
 $\langle \text{proof} \rangle$

lemma *prod-contI*:
assumes $f1: \bigwedge y. \text{cont } (\lambda x. f (x, y))$
assumes $f2: \bigwedge x. \text{cont } (\lambda y. f (x, y))$
shows $\text{cont } f$
 $\langle \text{proof} \rangle$

lemma *prod-cont-iff*: $\text{cont } f \longleftrightarrow (\forall y. \text{cont } (\lambda x. f (x, y))) \wedge (\forall x. \text{cont } (\lambda y. f (x, y)))$
 $\langle \text{proof} \rangle$

lemma *cont2cont-case-prod' [simp, cont2cont]*:
assumes $f: \text{cont } (\lambda p. f (\text{fst } p) (\text{fst } (\text{snd } p)) (\text{snd } (\text{snd } p)))$
assumes $g: \text{cont } (\lambda x. g x)$
shows $\text{cont } (\lambda x. \text{case-prod } (f x) (g x))$
 $\langle \text{proof} \rangle$

The simple version (due to Joachim Breitner) is needed if either element type of the pair is not a cpo.

lemma *cont2cont-split-simple [simp, cont2cont]*:
assumes $\bigwedge a b. \text{cont } (\lambda x. f x a b)$
shows $\text{cont } (\lambda x. \text{case } p \text{ of } (a, b) \Rightarrow f x a b)$
 $\langle \text{proof} \rangle$

Admissibility of predicates on product types.

lemma *adm-case-prod [simp]*:
assumes $\text{adm } (\lambda x. P x (\text{fst } (f x)) (\text{snd } (f x)))$
shows $\text{adm } (\lambda x. \text{case } f x \text{ of } (a, b) \Rightarrow P x a b)$
 $\langle \text{proof} \rangle$

7.7 Compactness and chain-finiteness

lemma *fst-below-iff*: $\text{fst } x \sqsubseteq y \longleftrightarrow x \sqsubseteq (y, \text{snd } x)$
for $x :: 'a \times 'b$
 $\langle \text{proof} \rangle$

lemma *snd-below-iff*: $\text{snd } x \sqsubseteq y \longleftrightarrow x \sqsubseteq (\text{fst } x, y)$
for $x :: 'a \times 'b$
 $\langle \text{proof} \rangle$

lemma *compact-fst*: $\text{compact } x \Longrightarrow \text{compact } (\text{fst } x)$
 $\langle \text{proof} \rangle$

lemma *compact-snd*: $compact\ x \implies compact\ (snd\ x)$
 ⟨*proof*⟩

lemma *compact-Pair*: $compact\ x \implies compact\ y \implies compact\ (x, y)$
 ⟨*proof*⟩

lemma *compact-Pair-iff* [*simp*]: $compact\ (x, y) \iff compact\ x \wedge compact\ y$
 ⟨*proof*⟩

instance *prod* :: $(chfin, chfin)\ chfin$
 ⟨*proof*⟩

end

8 The type of continuous functions

theory *Cfun*
imports *Cpodef Fun-Cpo Product-Cpo*
begin

default-sort *cpo*

8.1 Definition of continuous function type

definition *cfun* = $\{f :: 'a \Rightarrow 'b.\ cont\ f\}$

cpodef $('a, 'b)\ cfun\ ((- \rightarrow / -)\ [1, 0]\ 0) = cfun :: ('a \Rightarrow 'b)\ set$
 ⟨*proof*⟩

type-notation (*ASCII*)
cfun (**infixr** $->$ 0)

notation (*ASCII*)
Rep-cfun $((-\$/-)\ [999, 1000]\ 999)$

notation
Rep-cfun $((-./-)\ [999, 1000]\ 999)$

8.2 Syntax for continuous lambda abstraction

syntax *-cabs* :: $[logic, logic] \Rightarrow logic$

⟨*ML*⟩

Syntax for nested abstractions

syntax (*ASCII*)
-Lambda :: $[cargs, logic] \Rightarrow logic\ ((\exists LAM\ -./-)\ [1000, 10]\ 10)$

syntax

-Lambda :: [cargs, logic] \Rightarrow logic (($\exists \Lambda$ -./ -) [1000, 10] 10)

$\langle ML \rangle$

Dummy patterns for continuous abstraction

translations

Λ -. t \rightarrow CONST Abs-cfun (λ -. t)

8.3 Continuous function space is pointed

lemma bottom-cfun: $\perp \in$ cfun

$\langle proof \rangle$

instance cfun :: (cpo, discrete-cpo) discrete-cpo

$\langle proof \rangle$

instance cfun :: (cpo, pcpo) pcpo

$\langle proof \rangle$

lemmas Rep-cfun-strict =

typedef-Rep-strict [OF type-definition-cfun below-cfun-def bottom-cfun]

lemmas Abs-cfun-strict =

typedef-Abs-strict [OF type-definition-cfun below-cfun-def bottom-cfun]

function application is strict in its first argument

lemma Rep-cfun-strict1 [simp]: $\perp \cdot x = \perp$

$\langle proof \rangle$

lemma LAM-strict [simp]: $(\Lambda x. \perp) = \perp$

$\langle proof \rangle$

for compatibility with old HOLCF-Version

lemma inst-cfun-pcpo: $\perp = (\Lambda x. \perp)$

$\langle proof \rangle$

8.4 Basic properties of continuous functions

Beta-equality for continuous functions

lemma Abs-cfun-inverse2: cont f \Longrightarrow Rep-cfun (Abs-cfun f) = f

$\langle proof \rangle$

lemma beta-cfun: cont f \Longrightarrow $(\Lambda x. f x) \cdot u = f u$

$\langle proof \rangle$

8.4.1 Beta-reduction simproc

Given the term $(\Lambda x. f x) \cdot y$, the procedure tries to construct the theorem $(\Lambda x. f x) \cdot y \equiv f y$. If this theorem cannot be completely solved by the `cont2cont` rules, then the procedure returns the ordinary conditional *beta-cfun* rule.

The `simproc` does not solve any more goals that would be solved by using *beta-cfun* as a `simp` rule. The advantage of the `simproc` is that it can avoid deeply-nested calls to the simplifier that would otherwise be caused by large continuity side conditions.

Update: The `simproc` now uses rule *Abs-cfun-inverse2* instead of *beta-cfun*, to avoid problems with eta-contraction.

$\langle ML \rangle$

Eta-equality for continuous functions

lemma *eta-cfun*: $(\Lambda x. f \cdot x) = f$
 $\langle proof \rangle$

Extensionality for continuous functions

lemma *cfun-eq-iff*: $f = g \longleftrightarrow (\forall x. f \cdot x = g \cdot x)$
 $\langle proof \rangle$

lemma *cfun-eqI*: $(\bigwedge x. f \cdot x = g \cdot x) \Longrightarrow f = g$
 $\langle proof \rangle$

Extensionality wrt. ordering for continuous functions

lemma *cfun-below-iff*: $f \sqsubseteq g \longleftrightarrow (\forall x. f \cdot x \sqsubseteq g \cdot x)$
 $\langle proof \rangle$

lemma *cfun-belowI*: $(\bigwedge x. f \cdot x \sqsubseteq g \cdot x) \Longrightarrow f \sqsubseteq g$
 $\langle proof \rangle$

Congruence for continuous function application

lemma *cfun-cong*: $f = g \Longrightarrow x = y \Longrightarrow f \cdot x = g \cdot y$
 $\langle proof \rangle$

lemma *cfun-fun-cong*: $f = g \Longrightarrow f \cdot x = g \cdot x$
 $\langle proof \rangle$

lemma *cfun-arg-cong*: $x = y \Longrightarrow f \cdot x = f \cdot y$
 $\langle proof \rangle$

8.5 Continuity of application

lemma *cont-Rep-cfun1*: $cont (\lambda f. f \cdot x)$
 $\langle proof \rangle$

lemma *cont-Rep-cfun2*: $cont (\lambda x. f \cdot x)$

<proof>

lemmas *monofun-Rep-cfun = cont-Rep-cfun* [THEN cont2mono]

lemmas *monofun-Rep-cfun1 = cont-Rep-cfun1* [THEN cont2mono]

lemmas *monofun-Rep-cfun2 = cont-Rep-cfun2* [THEN cont2mono]

contlub, cont properties of *Rep-cfun* in each argument

lemma *contlub-cfun-arg: chain Y \implies f.(\sqcup i. Y i) = (\sqcup i. f.(Y i))*
<proof>

lemma *contlub-cfun-fun: chain F \implies (\sqcup i. F i)·x = (\sqcup i. F i·x)*
<proof>

monotonicity of application

lemma *monofun-cfun-fun: f \sqsubseteq g \implies f·x \sqsubseteq g·x*
<proof>

lemma *monofun-cfun-arg: x \sqsubseteq y \implies f·x \sqsubseteq f·y*
<proof>

lemma *monofun-cfun: f \sqsubseteq g \implies x \sqsubseteq y \implies f·x \sqsubseteq g·y*
<proof>

ch2ch - rules for the type *'a \rightarrow 'b*

lemma *chain-monofun: chain Y \implies chain (λ i. f.(Y i))*
<proof>

lemma *ch2ch-Rep-cfunR: chain Y \implies chain (λ i. f.(Y i))*
<proof>

lemma *ch2ch-Rep-cfunL: chain F \implies chain (λ i. (F i)·x)*
<proof>

lemma *ch2ch-Rep-cfun [simp]: chain F \implies chain Y \implies chain (λ i. (F i)·(Y i))*
<proof>

lemma *ch2ch-LAM [simp]:*
 $(\bigwedge x. \text{chain } (\lambda i. S i x)) \implies (\bigwedge i. \text{cont } (\lambda x. S i x)) \implies \text{chain } (\lambda i. \bigwedge x. S i x)$
<proof>

contlub, cont properties of *Rep-cfun* in both arguments

lemma *lub-APP: chain F \implies chain Y \implies (\sqcup i. F i·(Y i)) = (\sqcup i. F i)·(\sqcup i. Y i)*
<proof>

lemma *lub-LAM:*
assumes $\bigwedge x. \text{chain } (\lambda i. F i x)$

and $\bigwedge i. \text{cont } (\lambda x. F i x)$
shows $(\bigsqcup i. \Lambda x. F i x) = (\Lambda x. \bigsqcup i. F i x)$
 $\langle \text{proof} \rangle$

lemmas *lub-distrib* = *lub-APP* *lub-LAM*

strictness

lemma *strictI*: $f \cdot x = \perp \implies f \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

type $'a \rightarrow 'b$ is chain complete

lemma *lub-cfun*: $\text{chain } F \implies (\bigsqcup i. F i) = (\Lambda x. \bigsqcup i. F i \cdot x)$
 $\langle \text{proof} \rangle$

8.6 Continuity simplification procedure

cont2cont lemma for *Rep-cfun*

lemma *cont2cont-APP* [*simp*, *cont2cont*]:
assumes $f: \text{cont } (\lambda x. f x)$
assumes $t: \text{cont } (\lambda x. t x)$
shows $\text{cont } (\lambda x. (f x) \cdot (t x))$
 $\langle \text{proof} \rangle$

Two specific lemmas for the combination of LCF and HOL terms. These lemmas are needed in theories that use types like $'a \rightarrow 'b \Rightarrow 'c$.

lemma *cont-APP-app* [*simp*]: $\text{cont } f \implies \text{cont } g \implies \text{cont } (\lambda x. ((f x) \cdot (g x)) s)$
 $\langle \text{proof} \rangle$

lemma *cont-APP-app-app* [*simp*]: $\text{cont } f \implies \text{cont } g \implies \text{cont } (\lambda x. ((f x) \cdot (g x)) s t)$
 $\langle \text{proof} \rangle$

cont2mono Lemma for $\lambda x. \Lambda y. c1 x y$

lemma *cont2mono-LAM*:
 $\llbracket \bigwedge x. \text{cont } (\lambda y. f x y); \bigwedge y. \text{monofun } (\lambda x. f x y) \rrbracket$
 $\implies \text{monofun } (\lambda x. \Lambda y. f x y)$
 $\langle \text{proof} \rangle$

cont2cont Lemma for $\lambda x. \Lambda y. f x y$

Not suitable as a cont2cont rule, because on nested lambdas it causes exponential blow-up in the number of subgoals.

lemma *cont2cont-LAM*:
assumes $f1: \bigwedge x. \text{cont } (\lambda y. f x y)$
assumes $f2: \bigwedge y. \text{cont } (\lambda x. f x y)$
shows $\text{cont } (\lambda x. \Lambda y. f x y)$
 $\langle \text{proof} \rangle$

This version does work as a `cont2cont` rule, since it has only a single subgoal.

lemma *cont2cont-LAM'* [*simp*, *cont2cont*]:

fixes $f :: 'a::cpo \Rightarrow 'b::cpo \Rightarrow 'c::cpo$
assumes $f: cont (\lambda p. f (fst p) (snd p))$
shows $cont (\lambda x. \Lambda y. f x y)$
 $\langle proof \rangle$

lemma *cont2cont-LAM-discrete* [*simp*, *cont2cont*]:

$(\bigwedge y::'a::discrete-cpo. cont (\lambda x. f x y)) \Longrightarrow cont (\lambda x. \Lambda y. f x y)$
 $\langle proof \rangle$

8.7 Miscellaneous

Monotonicity of *Abs-cfun*

lemma *monofun-LAM*: $cont f \Longrightarrow cont g \Longrightarrow (\bigwedge x. f x \sqsubseteq g x) \Longrightarrow (\Lambda x. f x) \sqsubseteq (\Lambda x. g x)$
 $\langle proof \rangle$

some lemmata for functions with `flat`/`chfin` domain/range types

lemma *chfin-Rep-cfunR*: $chain Y \Longrightarrow \forall s. \exists n. (LUB i. Y i) \cdot s = Y n \cdot s$
for $Y :: nat \Rightarrow 'a::cpo \rightarrow 'b::chfin$
 $\langle proof \rangle$

lemma *adm-chfindom*: $adm (\lambda (u::'a::cpo \rightarrow 'b::chfin). P(u \cdot s))$
 $\langle proof \rangle$

8.8 Continuous injection-retraction pairs

Continuous retractions are strict.

lemma *retraction-strict*: $\forall x. f \cdot (g \cdot x) = x \Longrightarrow f \cdot \perp = \perp$
 $\langle proof \rangle$

lemma *injection-eq*: $\forall x. f \cdot (g \cdot x) = x \Longrightarrow (g \cdot x = g \cdot y) = (x = y)$
 $\langle proof \rangle$

lemma *injection-below*: $\forall x. f \cdot (g \cdot x) = x \Longrightarrow (g \cdot x \sqsubseteq g \cdot y) = (x \sqsubseteq y)$
 $\langle proof \rangle$

lemma *injection-defined-rev*: $\forall x. f \cdot (g \cdot x) = x \Longrightarrow g \cdot z = \perp \Longrightarrow z = \perp$
 $\langle proof \rangle$

lemma *injection-defined*: $\forall x. f \cdot (g \cdot x) = x \Longrightarrow z \neq \perp \Longrightarrow g \cdot z \neq \perp$
 $\langle proof \rangle$

a result about functions with `flat` codomain

lemma *flat-eqI*: $x \sqsubseteq y \Longrightarrow x \neq \perp \Longrightarrow x = y$
for $x y :: 'a::flat$

<proof>

lemma *flat-codom*: $f \cdot x = c \implies f \cdot \perp = \perp \vee (\forall z. f \cdot z = c)$
for $c :: 'b::flat$
<proof>

8.9 Identity and composition

definition *ID* :: $'a \rightarrow 'a$
where $ID = (\lambda x. x)$

definition *cfcomp* :: $('b \rightarrow 'c) \rightarrow ('a \rightarrow 'b) \rightarrow 'a \rightarrow 'c$
where *oo-def*: $cfcomp = (\lambda f g x. f \cdot (g \cdot x))$

abbreviation *cfcomp-syn* :: $['b \rightarrow 'c, 'a \rightarrow 'b] \Rightarrow 'a \rightarrow 'c$ (**infixr** *oo* 100)
where $f \text{ oo } g == cfcomp \cdot f \cdot g$

lemma *ID1* [*simp*]: $ID \cdot x = x$
<proof>

lemma *cfcomp1*: $(f \text{ oo } g) = (\lambda x. f \cdot (g \cdot x))$
<proof>

lemma *cfcomp2* [*simp*]: $(f \text{ oo } g) \cdot x = f \cdot (g \cdot x)$
<proof>

lemma *cfcomp-LAM*: $cont\ g \implies f \text{ oo } (\lambda x. g\ x) = (\lambda x. f \cdot (g\ x))$
<proof>

lemma *cfcomp-strict* [*simp*]: $\perp \text{ oo } f = \perp$
<proof>

Show that interpretation of $(pcpo, \dashrightarrow)$ is a category.

- The class of objects is interpretation of syntactical class *pcpo*.
- The class of arrows between objects $'a$ and $'b$ is interpret. of $'a \rightarrow 'b$.
- The identity arrow is interpretation of *ID*.
- The composition of *f* and *g* is interpretation of *oo*.

lemma *ID2* [*simp*]: $f \text{ oo } ID = f$
<proof>

lemma *ID3* [*simp*]: $ID \text{ oo } f = f$
<proof>

lemma *assoc-oo*: $f \text{ oo } (g \text{ oo } h) = (f \text{ oo } g) \text{ oo } h$
<proof>

8.10 Strictified functions

default-sort *pcpo*

definition *seq* :: 'a → 'b → 'b
 where *seq* = (λ x. if x = ⊥ then ⊥ else ID)

lemma *cont2cont-if-bottom* [*cont2cont*, *simp*]:
 assumes *f*: *cont* (λx. *f* x)
 and *g*: *cont* (λx. *g* x)
 shows *cont* (λx. if *f* x = ⊥ then ⊥ else *g* x)
 ⟨*proof*⟩

lemma *seq-conv-if*: *seq*·*x* = (if *x* = ⊥ then ⊥ else ID)
 ⟨*proof*⟩

lemma *seq-simps* [*simp*]:
seq·⊥ = ⊥
seq·*x*·⊥ = ⊥
x ≠ ⊥ ⇒ *seq*·*x* = ID
 ⟨*proof*⟩

definition *strictify* :: ('a → 'b) → 'a → 'b
 where *strictify* = (λ *f* x. *seq*·*x*·(*f*·*x*))

lemma *strictify-conv-if*: *strictify*·*f*·*x* = (if *x* = ⊥ then ⊥ else *f*·*x*)
 ⟨*proof*⟩

lemma *strictify1* [*simp*]: *strictify*·*f*·⊥ = ⊥
 ⟨*proof*⟩

lemma *strictify2* [*simp*]: *x* ≠ ⊥ ⇒ *strictify*·*f*·*x* = *f*·*x*
 ⟨*proof*⟩

8.11 Continuity of let-bindings

lemma *cont2cont-Let*:
 assumes *f*: *cont* (λx. *f* x)
 assumes *g1*: ∧*y*. *cont* (λx. *g* x y)
 assumes *g2*: ∧*x*. *cont* (λy. *g* x y)
 shows *cont* (λx. let *y* = *f* x in *g* x y)
 ⟨*proof*⟩

lemma *cont2cont-Let'* [*simp*, *cont2cont*]:
 assumes *f*: *cont* (λx. *f* x)
 assumes *g*: *cont* (λp. *g* (fst p) (snd p))
 shows *cont* (λx. let *y* = *f* x in *g* x y)
 ⟨*proof*⟩

The simple version (suggested by Joachim Breitner) is needed if the type of

the defined term is not a cpo.

lemma *cont2cont-Let-simple* [*simp*, *cont2cont*]:
assumes $\bigwedge y. \text{cont } (\lambda x. g \ x \ y)$
shows $\text{cont } (\lambda x. \text{let } y = t \ \text{in } g \ x \ y)$
 $\langle \text{proof} \rangle$

end

9 Continuous deflations and ep-pairs

theory *Deflation*
imports *Cfun*
begin

default-sort *cpo*

9.1 Continuous deflations

locale *deflation* =
fixes $d :: 'a \rightarrow 'a$
assumes *idem*: $\bigwedge x. d \cdot (d \cdot x) = d \cdot x$
assumes *below*: $\bigwedge x. d \cdot x \sqsubseteq x$
begin

lemma *below-ID*: $d \sqsubseteq ID$
 $\langle \text{proof} \rangle$

The set of fixed points is the same as the range.

lemma *fixes-eq-range*: $\{x. d \cdot x = x\} = \text{range } (\lambda x. d \cdot x)$
 $\langle \text{proof} \rangle$

lemma *range-eq-fixes*: $\text{range } (\lambda x. d \cdot x) = \{x. d \cdot x = x\}$
 $\langle \text{proof} \rangle$

The pointwise ordering on deflation functions coincides with the subset ordering of their sets of fixed-points.

lemma *belowI*:
assumes $f: \bigwedge x. d \cdot x = x \implies f \cdot x = x$
shows $d \sqsubseteq f$
 $\langle \text{proof} \rangle$

lemma *belowD*: $\llbracket f \sqsubseteq d; f \cdot x = x \rrbracket \implies d \cdot x = x$
 $\langle \text{proof} \rangle$

end

lemma *deflation-strict*: $\text{deflation } d \implies d \cdot \perp = \perp$
 $\langle \text{proof} \rangle$

lemma *adm-deflation*: *adm* ($\lambda d.$ *deflation* d)
 ⟨*proof*⟩

lemma *deflation-ID*: *deflation* ID
 ⟨*proof*⟩

lemma *deflation-bottom*: *deflation* \perp
 ⟨*proof*⟩

lemma *deflation-below-iff*: *deflation* $p \implies$ *deflation* $q \implies p \sqsubseteq q \iff (\forall x. p \cdot x = x \longrightarrow q \cdot x = x)$
 ⟨*proof*⟩

The composition of two deflations is equal to the lesser of the two (if they are comparable).

lemma *deflation-below-comp1*:
assumes *deflation* f
assumes *deflation* g
shows $f \sqsubseteq g \implies f \cdot (g \cdot x) = f \cdot x$
 ⟨*proof*⟩

lemma *deflation-below-comp2*: *deflation* $f \implies$ *deflation* $g \implies f \sqsubseteq g \implies g \cdot (f \cdot x) = f \cdot x$
 ⟨*proof*⟩

9.2 Deflations with finite range

lemma *finite-range-imp-finite-fixes*:
assumes *finite* (*range* f)
shows *finite* $\{x. f \cdot x = x\}$
 ⟨*proof*⟩

locale *finite-deflation* = *deflation* +
assumes *finite-fixes*: *finite* $\{x. d \cdot x = x\}$
begin

lemma *finite-range*: *finite* (*range* $(\lambda x. d \cdot x)$)
 ⟨*proof*⟩

lemma *finite-image*: *finite* $((\lambda x. d \cdot x) \text{ ‘ } A)$
 ⟨*proof*⟩

lemma *compact*: *compact* $(d \cdot x)$
 ⟨*proof*⟩

end

lemma *finite-deflation-intro*: *deflation* $d \implies$ *finite* $\{x. d \cdot x = x\} \implies$ *finite-deflation*

d
 ⟨proof⟩

lemma *finite-deflation-imp-deflation*: *finite-deflation* $d \implies$ *deflation* d
 ⟨proof⟩

lemma *finite-deflation-bottom*: *finite-deflation* \perp
 ⟨proof⟩

9.3 Continuous embedding-projection pairs

locale *ep-pair* =
fixes $e :: 'a \rightarrow 'b$ **and** $p :: 'b \rightarrow 'a$
assumes *e-inverse* [*simp*]: $\bigwedge x. p \cdot (e \cdot x) = x$
and *e-p-below*: $\bigwedge y. e \cdot (p \cdot y) \sqsubseteq y$
begin

lemma *e-below-iff* [*simp*]: $e \cdot x \sqsubseteq e \cdot y \longleftrightarrow x \sqsubseteq y$
 ⟨proof⟩

lemma *e-eq-iff* [*simp*]: $e \cdot x = e \cdot y \longleftrightarrow x = y$
 ⟨proof⟩

lemma *p-eq-iff*: $e \cdot (p \cdot x) = x \implies e \cdot (p \cdot y) = y \implies p \cdot x = p \cdot y \longleftrightarrow x = y$
 ⟨proof⟩

lemma *p-inverse*: $(\exists x. y = e \cdot x) \longleftrightarrow e \cdot (p \cdot y) = y$
 ⟨proof⟩

lemma *e-below-iff-below-p*: $e \cdot x \sqsubseteq y \longleftrightarrow x \sqsubseteq p \cdot y$
 ⟨proof⟩

lemma *compact-e-rev*: *compact* $(e \cdot x) \implies$ *compact* x
 ⟨proof⟩

lemma *compact-e*:
assumes *compact* x
shows *compact* $(e \cdot x)$
 ⟨proof⟩

lemma *compact-e-iff*: *compact* $(e \cdot x) \longleftrightarrow$ *compact* x
 ⟨proof⟩

Deflations from ep-pairs

lemma *deflation-e-p*: *deflation* $(e \circ p)$
 ⟨proof⟩

lemma *deflation-e-d-p*:
assumes *deflation* d

shows *deflation* ($e \circ d \circ p$)
 ⟨*proof*⟩

lemma *finite-deflation-e-d-p*:
assumes *finite-deflation* d
shows *finite-deflation* ($e \circ d \circ p$)
 ⟨*proof*⟩

lemma *deflation-p-d-e*:
assumes *deflation* d
assumes $d: \bigwedge x. d \cdot x \sqsubseteq e \cdot (p \cdot x)$
shows *deflation* ($p \circ d \circ e$)
 ⟨*proof*⟩

lemma *finite-deflation-p-d-e*:
assumes *finite-deflation* d
assumes $d: \bigwedge x. d \cdot x \sqsubseteq e \cdot (p \cdot x)$
shows *finite-deflation* ($p \circ d \circ e$)
 ⟨*proof*⟩

end

9.4 Uniqueness of ep-pairs

lemma *ep-pair-unique-e-lemma*:
assumes $1: \text{ep-pair } e1 \ p$
and $2: \text{ep-pair } e2 \ p$
shows $e1 \sqsubseteq e2$
 ⟨*proof*⟩

lemma *ep-pair-unique-e*: $\text{ep-pair } e1 \ p \implies \text{ep-pair } e2 \ p \implies e1 = e2$
 ⟨*proof*⟩

lemma *ep-pair-unique-p-lemma*:
assumes $1: \text{ep-pair } e \ p1$
and $2: \text{ep-pair } e \ p2$
shows $p1 \sqsubseteq p2$
 ⟨*proof*⟩

lemma *ep-pair-unique-p*: $\text{ep-pair } e \ p1 \implies \text{ep-pair } e \ p2 \implies p1 = p2$
 ⟨*proof*⟩

9.5 Composing ep-pairs

lemma *ep-pair-ID-ID*: $\text{ep-pair } ID \ ID$
 ⟨*proof*⟩

lemma *ep-pair-comp*:
assumes $\text{ep-pair } e1 \ p1$ **and** $\text{ep-pair } e2 \ p2$
shows $\text{ep-pair } (e2 \circ e1) \ (p1 \circ p2)$

<proof>

locale *pcpo-ep-pair* = *ep-pair e p*
for $e :: 'a::pcpo \rightarrow 'b::pcpo$
and $p :: 'b::pcpo \rightarrow 'a::pcpo$
begin

lemma *e-strict* [*simp*]: $e.\perp = \perp$
<proof>

lemma *e-bottom-iff* [*simp*]: $e.x = \perp \longleftrightarrow x = \perp$
<proof>

lemma *e-defined*: $x \neq \perp \implies e.x \neq \perp$
<proof>

lemma *p-strict* [*simp*]: $p.\perp = \perp$
<proof>

lemmas *stricts* = *e-strict p-strict*

end

end

10 The type of strict products

theory *Sprod*
imports *Cfun*
begin

default-sort *pcpo*

10.1 Definition of strict product type

definition *sprod* = $\{p::'a \times 'b. p = \perp \vee (fst\ p \neq \perp \wedge snd\ p \neq \perp)\}$

pcpodef ($'a, 'b$) *sprod* (($- \otimes / -$) [*21,20*] *20*) = *sprod* :: ($'a \times 'b$) *set*
<proof>

instance *sprod* :: ($\{chfin,pcpo\}, \{chfin,pcpo\}$) *chfin*
<proof>

type-notation (*ASCII*)
sprod (**infixr** ****** *20*)

10.2 Definitions of constants

definition *sfst* :: ($'a ** 'b$) $\rightarrow 'a$

where $sfst = (\Lambda p. fst (Rep-sprod\ p))$

definition $ssnd :: ('a ** 'b) \rightarrow 'b$
where $ssnd = (\Lambda p. snd (Rep-sprod\ p))$

definition $spair :: 'a \rightarrow 'b \rightarrow ('a ** 'b)$
where $spair = (\Lambda a\ b. Abs-sprod (seq\cdot b\cdot a, seq\cdot a\cdot b))$

definition $ssplit :: ('a \rightarrow 'b \rightarrow 'c) \rightarrow ('a ** 'b) \rightarrow 'c$
where $ssplit = (\Lambda f\ p. seq\cdot p\cdot (f\cdot (sfst\cdot p)\cdot (ssnd\cdot p)))$

syntax $-stuple :: [logic, args] \Rightarrow logic\ ((1'(-:/ -:'))$

translations

$(:x, y, z:) \Leftrightarrow (:x, (:y, z):)$
 $(:x, y:) \Leftrightarrow CONST\ spair\cdot x\cdot y$

translations

$\Lambda(CONST\ spair\cdot x\cdot y). t \Leftrightarrow CONST\ ssplit\cdot(\Lambda\ x\ y. t)$

10.3 Case analysis

lemma $spair-sprod: (seq\cdot b\cdot a, seq\cdot a\cdot b) \in sprod$
 $\langle proof \rangle$

lemma $Rep-sprod-spair: Rep-sprod\ (:a, b:) = (seq\cdot b\cdot a, seq\cdot a\cdot b)$
 $\langle proof \rangle$

lemmas $Rep-sprod-simps =$
 $Rep-sprod-inject\ [symmetric]\ below-sprod-def$
 $prod-eq-iff\ below-prod-def$
 $Rep-sprod-strict\ Rep-sprod-spair$

lemma $sprodE\ [case-names\ bottom\ spair, cases\ type: sprod]:$
obtains $p = \perp \mid x\ y$ **where** $p = (:x, y:)$ **and** $x \neq \perp$ **and** $y \neq \perp$
 $\langle proof \rangle$

lemma $sprod-induct\ [case-names\ bottom\ spair, induct\ type: sprod]:$
 $\llbracket P\ \perp; \bigwedge x\ y. \llbracket x \neq \perp; y \neq \perp \rrbracket \implies P\ (:x, y:) \rrbracket \implies P\ x$
 $\langle proof \rangle$

10.4 Properties of *spair*

lemma $spair-strict1\ [simp]: (:\perp, y:) = \perp$
 $\langle proof \rangle$

lemma $spair-strict2\ [simp]: (:x, \perp:) = \perp$
 $\langle proof \rangle$

lemma $spair-bottom-iff\ [simp]: (:x, y:) = \perp \iff x = \perp \vee y = \perp$
 $\langle proof \rangle$

lemma *spair-below-iff*: $(:a, b:) \sqsubseteq (:c, d:) \iff a = \perp \vee b = \perp \vee (a \sqsubseteq c \wedge b \sqsubseteq d)$
 ⟨proof⟩

lemma *spair-eq-iff*: $(:a, b:) = (:c, d:) \iff a = c \wedge b = d \vee (a = \perp \vee b = \perp) \wedge (c = \perp \vee d = \perp)$
 ⟨proof⟩

lemma *spair-strict*: $x = \perp \vee y = \perp \implies (:x, y:) = \perp$
 ⟨proof⟩

lemma *spair-strict-rev*: $(:x, y:) \neq \perp \implies x \neq \perp \wedge y \neq \perp$
 ⟨proof⟩

lemma *spair-defined*: $\llbracket x \neq \perp; y \neq \perp \rrbracket \implies (:x, y:) \neq \perp$
 ⟨proof⟩

lemma *spair-defined-rev*: $(:x, y:) = \perp \implies x = \perp \vee y = \perp$
 ⟨proof⟩

lemma *spair-below*: $x \neq \perp \implies y \neq \perp \implies (:x, y:) \sqsubseteq (:a, b:) \iff x \sqsubseteq a \wedge y \sqsubseteq b$
 ⟨proof⟩

lemma *spair-eq*: $x \neq \perp \implies y \neq \perp \implies (:x, y:) = (:a, b:) \iff x = a \wedge y = b$
 ⟨proof⟩

lemma *spair-inject*: $x \neq \perp \implies y \neq \perp \implies (:x, y:) = (:a, b:) \implies x = a \wedge y = b$
 ⟨proof⟩

lemma *inst-sprod-pcpo2*: $\perp = (:\perp, \perp:)$
 ⟨proof⟩

lemma *sprodE2*: $(\bigwedge x y. p = (:x, y:) \implies Q) \implies Q$
 ⟨proof⟩

10.5 Properties of *sfst* and *ssnd*

lemma *sfst-strict* [*simp*]: $sfst.\perp = \perp$
 ⟨proof⟩

lemma *ssnd-strict* [*simp*]: $ssnd.\perp = \perp$
 ⟨proof⟩

lemma *sfst-spair* [*simp*]: $y \neq \perp \implies sfst.(:x, y:) = x$
 ⟨proof⟩

lemma *ssnd-spair* [*simp*]: $x \neq \perp \implies ssnd.(:x, y:) = y$
 ⟨proof⟩

lemma *sfst-bottom-iff* [*simp*]: $sfst.p = \perp \longleftrightarrow p = \perp$
 ⟨*proof*⟩

lemma *ssnd-bottom-iff* [*simp*]: $ssnd.p = \perp \longleftrightarrow p = \perp$
 ⟨*proof*⟩

lemma *sfst-defined*: $p \neq \perp \implies sfst.p \neq \perp$
 ⟨*proof*⟩

lemma *ssnd-defined*: $p \neq \perp \implies ssnd.p \neq \perp$
 ⟨*proof*⟩

lemma *spair-sfst-ssnd*: $(:sfst.p, ssnd.p:) = p$
 ⟨*proof*⟩

lemma *below-sprod*: $x \sqsubseteq y \longleftrightarrow sfst.x \sqsubseteq sfst.y \wedge ssnd.x \sqsubseteq ssnd.y$
 ⟨*proof*⟩

lemma *eq-sprod*: $x = y \longleftrightarrow sfst.x = sfst.y \wedge ssnd.x = ssnd.y$
 ⟨*proof*⟩

lemma *sfst-below-iff*: $sfst.x \sqsubseteq y \longleftrightarrow x \sqsubseteq (:y, ssnd.x:)$
 ⟨*proof*⟩

lemma *ssnd-below-iff*: $ssnd.x \sqsubseteq y \longleftrightarrow x \sqsubseteq (:sfst.x, y:)$
 ⟨*proof*⟩

10.6 Compactness

lemma *compact-sfst*: $compact\ x \implies compact\ (sfst.x)$
 ⟨*proof*⟩

lemma *compact-ssnd*: $compact\ x \implies compact\ (ssnd.x)$
 ⟨*proof*⟩

lemma *compact-spair*: $compact\ x \implies compact\ y \implies compact\ (:x, y:)$
 ⟨*proof*⟩

lemma *compact-spair-iff*: $compact\ (:x, y:) \longleftrightarrow x = \perp \vee y = \perp \vee (compact\ x \wedge compact\ y)$
 ⟨*proof*⟩

10.7 Properties of *ssplit*

lemma *ssplit1* [*simp*]: $ssplit.f.\perp = \perp$
 ⟨*proof*⟩

lemma *ssplit2* [*simp*]: $x \neq \perp \implies y \neq \perp \implies ssplit.f.(:x, y:) = f.x.y$
 ⟨*proof*⟩

lemma *ssplit3* [*simp*]: *ssplit*·*spair*·*z* = *z*
 ⟨*proof*⟩

10.8 Strict product preserves flatness

instance *sprod* :: (*flat*, *flat*) *flat*
 ⟨*proof*⟩

end

11 Discrete cpo types

theory *Discrete*
imports *Cont*
begin

datatype *'a discr* = *Discr 'a* :: *type*

11.1 Discrete cpo class instance

instantiation *discr* :: (*type*) *discrete-cpo*
begin

definition ((\sqsubseteq) :: *'a discr* \Rightarrow *'a discr* \Rightarrow *bool*) = (=)

instance
 ⟨*proof*⟩

end

11.2 *undiscr*

definition *undiscr* :: (*'a::type*)*discr* \Rightarrow *'a*
where *undiscr* *x* = (*case* *x* *of* *Discr* *y* \Rightarrow *y*)

lemma *undiscr-Discr* [*simp*]: *undiscr* (*Discr* *x*) = *x*
 ⟨*proof*⟩

lemma *Discr-undiscr* [*simp*]: *Discr* (*undiscr* *y*) = *y*
 ⟨*proof*⟩

end

12 The type of lifted values

theory *Up*
imports *Cfun*
begin

default-sort *cpo*

12.1 Definition of new type for lifting

datatype 'a u ((- \perp) [1000] 999) = *Ibottom* | *Iup* 'a

primrec *Ifup* :: ('a \rightarrow 'b::*pcpo*) \Rightarrow 'a u \Rightarrow 'b

where

Ifup *f* *Ibottom* = \perp
 | *Ifup* *f* (*Iup* *x*) = *f*·*x*

12.2 Ordering on lifted cpo

instantiation *u* :: (*cpo*) *below*

begin

definition *below-up-def*:

(\sqsubseteq) \equiv
 ($\lambda x y.$
 (case *x* of
 Ibottom \Rightarrow *True*
 | *Iup* *a* \Rightarrow (case *y* of *Ibottom* \Rightarrow *False* | *Iup* *b* \Rightarrow *a* \sqsubseteq *b*)))

instance \langle *proof* \rangle

end

lemma *minimal-up* [*iff*]: *Ibottom* \sqsubseteq *z*
 \langle *proof* \rangle

lemma *not-Iup-below* [*iff*]: *Iup* *x* $\not\sqsubseteq$ *Ibottom*
 \langle *proof* \rangle

lemma *Iup-below* [*iff*]: (*Iup* *x* \sqsubseteq *Iup* *y*) = (*x* \sqsubseteq *y*)
 \langle *proof* \rangle

12.3 Lifted cpo is a partial order

instance *u* :: (*cpo*) *po*
 \langle *proof* \rangle

12.4 Lifted cpo is a cpo

lemma *is-lub-Iup*: *range* *S* $\ll\langle$ *x* \Longrightarrow *range* ($\lambda i. Iup (S i)$) $\ll\langle$ *Iup* *x*
 \langle *proof* \rangle

lemma *up-chain-lemma*:

assumes *Y*: *chain* *Y*

obtains $\forall i. Y i = Ibottom$

| A k **where** $\forall i. Iup (A i) = Y (i + k)$ **and** *chain* A **and** *range* $Y \ll | Iup$
 $(\bigsqcup i. A i)$
 $\langle proof \rangle$

instance $u :: (cpo) cpo$
 $\langle proof \rangle$

12.5 Lifted cpo is pointed

instance $u :: (cpo) pcpo$
 $\langle proof \rangle$

for compatibility with old HOLCF-Version

lemma *inst-up-pcpo*: $\perp = Ibottom$
 $\langle proof \rangle$

12.6 Continuity of *Iup* and *Ifup*

continuity for *Iup*

lemma *cont-Iup*: *cont* *Iup*
 $\langle proof \rangle$

continuity for *Ifup*

lemma *cont-Ifup1*: *cont* $(\lambda f. Ifup f x)$
 $\langle proof \rangle$

lemma *monofun-Ifup2*: *monofun* $(\lambda x. Ifup f x)$
 $\langle proof \rangle$

lemma *cont-Ifup2*: *cont* $(\lambda x. Ifup f x)$
 $\langle proof \rangle$

12.7 Continuous versions of constants

definition $up :: 'a \rightarrow 'a$ u
where $up = (\Lambda x. Iup x)$

definition $fup :: ('a \rightarrow 'b::pcpo) \rightarrow 'a \rightarrow 'b$
where $fup = (\Lambda f p. Ifup f p)$

translations

case l *of* $XCONST up \cdot x \Rightarrow t \Leftrightarrow CONST fup \cdot (\Lambda x. t) \cdot l$
case l *of* $(XCONST up :: 'a) \cdot x \Rightarrow t \rightarrow CONST fup \cdot (\Lambda x. t) \cdot l$
 $\Lambda (XCONST up \cdot x). t \Leftrightarrow CONST fup \cdot (\Lambda x. t)$

continuous versions of lemmas for $'a_{\perp}$

lemma *Exh-Up*: $z = \perp \vee (\exists x. z = up \cdot x)$
 $\langle proof \rangle$

lemma *up-eq* [*simp*]: $(up \cdot x = up \cdot y) = (x = y)$
 ⟨*proof*⟩

lemma *up-inject*: $up \cdot x = up \cdot y \implies x = y$
 ⟨*proof*⟩

lemma *up-defined* [*simp*]: $up \cdot x \neq \perp$
 ⟨*proof*⟩

lemma *not-up-less-UU*: $up \cdot x \not\sqsubseteq \perp$
 ⟨*proof*⟩

lemma *up-below* [*simp*]: $up \cdot x \sqsubseteq up \cdot y \iff x \sqsubseteq y$
 ⟨*proof*⟩

lemma *upE* [*case-names bottom up, cases type: u*]: $\llbracket p = \perp \implies Q; \bigwedge x. p = up \cdot x \implies Q \rrbracket \implies Q$
 ⟨*proof*⟩

lemma *up-induct* [*case-names bottom up, induct type: u*]: $P \perp \implies (\bigwedge x. P (up \cdot x)) \implies P x$
 ⟨*proof*⟩

lifting preserves chain-finiteness

lemma *up-chain-cases*:

assumes Y : *chain* Y

obtains $\forall i. Y i = \perp$

| $A k$ **where** $\forall i. up \cdot (A i) = Y (i + k)$ **and** *chain* A **and** $(\bigsqcup i. Y i) = up \cdot (\bigsqcup i. A i)$

⟨*proof*⟩

lemma *compact-up*: $compact\ x \implies compact\ (up \cdot x)$
 ⟨*proof*⟩

lemma *compact-upD*: $compact\ (up \cdot x) \implies compact\ x$
 ⟨*proof*⟩

lemma *compact-up-iff* [*simp*]: $compact\ (up \cdot x) = compact\ x$
 ⟨*proof*⟩

instance $u :: (chfin)\ chfin$
 ⟨*proof*⟩

properties of fup

lemma *fup1* [*simp*]: $fup \cdot f \cdot \perp = \perp$
 ⟨*proof*⟩

lemma *fup2* [*simp*]: $fup \cdot f \cdot (up \cdot x) = f \cdot x$

<proof>

lemma *fup3* [*simp*]: $fup \cdot up \cdot x = x$
<proof>

end

13 Lifting types of class type to flat pcpo’s

theory *Lift*
imports *Discrete Up*
begin

default-sort *type*

pcpodef *'a lift* = *UNIV* :: *'a discr u set*
<proof>

lemmas *inst-lift-pcpo* = *Abs-lift-strict* [*symmetric*]

definition

Def :: *'a* \Rightarrow *'a lift* **where**
Def *x* = *Abs-lift* (*up* · (*Discr* *x*))

13.1 Lift as a datatype

lemma *lift-induct*: $\llbracket P \perp; \bigwedge x. P (Def\ x) \rrbracket \Longrightarrow P\ y$
<proof>

old-rep-datatype $\perp :: 'a\ lift\ Def$
<proof>

\perp and *Def*

lemma *not-Undef-is-Def*: $(x \neq \perp) = (\exists y. x = Def\ y)$
<proof>

lemma *lift-definedE*: $\llbracket x \neq \perp; \bigwedge a. x = Def\ a \Longrightarrow R \rrbracket \Longrightarrow R$
<proof>

For $x \neq \perp$ in assumptions *defined* replaces *x* by *Def a* in conclusion.

<ML>

lemma *DefE*: $Def\ x = \perp \Longrightarrow R$
<proof>

lemma *DefE2*: $\llbracket x = Def\ s; x = \perp \rrbracket \Longrightarrow R$
<proof>

lemma *Def-below-Def*: $Def\ x \sqsubseteq Def\ y \longleftrightarrow x = y$
 $\langle proof \rangle$

lemma *Def-below-iff [simp]*: $Def\ x \sqsubseteq y \longleftrightarrow Def\ x = y$
 $\langle proof \rangle$

13.2 Lift is flat

instance *lift* :: (type) flat
 $\langle proof \rangle$

13.3 Continuity of case-lift

lemma *case-lift-eq*: $case\ lift\ \perp\ f\ x = fup\ (\Lambda\ y.\ f\ (undiscr\ y))\ \cdot\ (Rep\ lift\ x)$
 $\langle proof \rangle$

lemma *cont2cont-case-lift [simp]*:
 $\llbracket \Lambda\ y.\ cont\ (\lambda x.\ f\ x\ y); cont\ g \rrbracket \implies cont\ (\lambda x.\ case\ lift\ \perp\ (f\ x)\ (g\ x))$
 $\langle proof \rangle$

13.4 Further operations

definition

flift1 :: ('a \Rightarrow 'b::pcpo) \Rightarrow ('a lift \rightarrow 'b) (**binder** FLIFT 10) **where**
 $flift1 = (\lambda f.\ (\Lambda\ x.\ case\ lift\ \perp\ f\ x))$

translations

$\Lambda(XCONST\ Def\ x).\ t \Rightarrow CONST\ flift1\ (\lambda x.\ t)$
 $\Lambda(CONST\ Def\ x).\ FLIFT\ y.\ t \leq FLIFT\ x\ y.\ t$
 $\Lambda(CONST\ Def\ x).\ t \leq FLIFT\ x.\ t$

definition

flift2 :: ('a \Rightarrow 'b) \Rightarrow ('a lift \rightarrow 'b lift) **where**
 $flift2\ f = (FLIFT\ x.\ Def\ (f\ x))$

lemma *flift1-Def [simp]*: $flift1\ f\ \cdot\ (Def\ x) = (f\ x)$
 $\langle proof \rangle$

lemma *flift2-Def [simp]*: $flift2\ f\ \cdot\ (Def\ x) = Def\ (f\ x)$
 $\langle proof \rangle$

lemma *flift1-strict [simp]*: $flift1\ f\ \cdot\ \perp = \perp$
 $\langle proof \rangle$

lemma *flift2-strict [simp]*: $flift2\ f\ \cdot\ \perp = \perp$
 $\langle proof \rangle$

lemma *flift2-defined [simp]*: $x \neq \perp \implies (flift2\ f)\ \cdot\ x \neq \perp$
 $\langle proof \rangle$

lemma *flift2-bottom-iff* [*simp*]: $(\text{flift2 } f \cdot x = \perp) = (x = \perp)$
 $\langle \text{proof} \rangle$

lemma *FLIFT-mono*:
 $(\bigwedge x. f x \sqsubseteq g x) \implies (\text{FLIFT } x. f x) \sqsubseteq (\text{FLIFT } x. g x)$
 $\langle \text{proof} \rangle$

lemma *cont2cont-flift1* [*simp, cont2cont*]:
 $\llbracket \bigwedge y. \text{cont } (\lambda x. f x y) \rrbracket \implies \text{cont } (\lambda x. \text{FLIFT } y. f x y)$
 $\langle \text{proof} \rangle$

end

14 The type of lifted booleans

theory *Tr*
imports *Lift*
begin

14.1 Type definition and constructors

type-synonym *tr* = *bool lift*

translations
 $(\text{type}) \text{ tr} \leftarrow (\text{type}) \text{ bool lift}$

definition *TT* :: *tr*
where *TT* = *Def True*

definition *FF* :: *tr*
where *FF* = *Def False*

Exhaustion and Elimination for type *tr*

lemma *Exh-tr*: $t = \perp \vee t = \text{TT} \vee t = \text{FF}$
 $\langle \text{proof} \rangle$

lemma *trE* [*case-names bottom TT FF, cases type: tr*]:
 $\llbracket p = \perp \implies Q; p = \text{TT} \implies Q; p = \text{FF} \implies Q \rrbracket \implies Q$
 $\langle \text{proof} \rangle$

lemma *tr-induct* [*case-names bottom TT FF, induct type: tr*]:
 $P \perp \implies P \text{TT} \implies P \text{FF} \implies P x$
 $\langle \text{proof} \rangle$

distinctness for type *tr*

lemma *dist-below-tr* [*simp*]:
 $\text{TT} \not\sqsubseteq \perp \text{FF} \not\sqsubseteq \perp \text{TT} \not\sqsubseteq \text{FF} \text{FF} \not\sqsubseteq \text{TT}$
 $\langle \text{proof} \rangle$

lemma *dist-eq-tr* [*simp*]: $TT \neq \perp \text{ } FF \neq \perp \text{ } TT \neq FF \perp \neq TT \perp \neq FF \text{ } FF \neq TT$
 ⟨*proof*⟩

lemma *TT-below-iff* [*simp*]: $TT \sqsubseteq x \longleftrightarrow x = TT$
 ⟨*proof*⟩

lemma *FF-below-iff* [*simp*]: $FF \sqsubseteq x \longleftrightarrow x = FF$
 ⟨*proof*⟩

lemma *not-below-TT-iff* [*simp*]: $x \not\sqsubseteq TT \longleftrightarrow x = FF$
 ⟨*proof*⟩

lemma *not-below-FF-iff* [*simp*]: $x \not\sqsubseteq FF \longleftrightarrow x = TT$
 ⟨*proof*⟩

14.2 Case analysis

default-sort *pcpo*

definition *tr-case* :: $'a \rightarrow 'a \rightarrow tr \rightarrow 'a$
 where *tr-case* = $(\Lambda t e (Def b). \text{if } b \text{ then } t \text{ else } e)$

abbreviation *cifte-syn* :: $[tr, 'c, 'c] \Rightarrow 'c$ ((*If* (-)/ *then* (-)/ *else* (-)) [0, 0, 60]
 60)
 where *If* *b then e1 else e2* $\equiv tr\text{-case}\cdot e1\cdot e2\cdot b$

translations

$\Lambda (XCONST TT). t \Rightarrow CONST tr\text{-case}\cdot t\cdot \perp$
 $\Lambda (XCONST FF). t \Rightarrow CONST tr\text{-case}\cdot \perp\cdot t$

lemma *ifte-thms* [*simp*]:
If \perp *then e1 else e2* = \perp
If FF *then e1 else e2* = *e2*
If TT *then e1 else e2* = *e1*
 ⟨*proof*⟩

14.3 Boolean connectives

definition *trand* :: $tr \rightarrow tr \rightarrow tr$
 where *andalso-def*: *trand* = $(\Lambda x y. \text{If } x \text{ then } y \text{ else } FF)$

abbreviation *andalso-syn* :: $tr \Rightarrow tr \Rightarrow tr$ (- *andalso* - [36,35] 35)
 where *x andalso y* $\equiv trand\cdot x\cdot y$

definition *tror* :: $tr \rightarrow tr \rightarrow tr$
 where *orelse-def*: *tror* = $(\Lambda x y. \text{If } x \text{ then } TT \text{ else } y)$

abbreviation *orelse-syn* :: $tr \Rightarrow tr \Rightarrow tr$ (- *orelse* - [31,30] 30)
 where *x orelse y* $\equiv tror\cdot x\cdot y$

definition $neg :: tr \rightarrow tr$

where $neg = flift2\ Not$

definition $If2 :: tr \Rightarrow 'c \Rightarrow 'c \Rightarrow 'c$

where $If2\ Q\ x\ y = (If\ Q\ then\ x\ else\ y)$

tactic for tr-thms with case split

lemmas $tr-defs = andalso-def\ orelse-def\ neg-def\ tr-case-def\ TT-def\ FF-def$

lemmas about andalso, orelse, neg and if

lemma $andalso-thms [simp]:$

$(TT\ andalso\ y) = y$
 $(FF\ andalso\ y) = FF$
 $(\perp\ andalso\ y) = \perp$
 $(y\ andalso\ TT) = y$
 $(y\ andalso\ y) = y$
 $\langle proof \rangle$

lemma $orelse-thms [simp]:$

$(TT\ orelse\ y) = TT$
 $(FF\ orelse\ y) = y$
 $(\perp\ orelse\ y) = \perp$
 $(y\ orelse\ FF) = y$
 $(y\ orelse\ y) = y$
 $\langle proof \rangle$

lemma $neg-thms [simp]:$

$neg \cdot TT = FF$
 $neg \cdot FF = TT$
 $neg \cdot \perp = \perp$
 $\langle proof \rangle$

split-tac for If via If2 because the constant has to be a constant

lemma $split-If2: P (If2\ Q\ x\ y) \longleftrightarrow ((Q = \perp \longrightarrow P\ \perp) \wedge (Q = TT \longrightarrow P\ x) \wedge (Q = FF \longrightarrow P\ y))$
 $\langle proof \rangle$

$\langle ML \rangle$

14.4 Rewriting of HOLCF operations to HOL functions

lemma $andalso-or: t \neq \perp \Longrightarrow (t\ andalso\ s) = FF \longleftrightarrow t = FF \vee s = FF$
 $\langle proof \rangle$

lemma $andalso-and: t \neq \perp \Longrightarrow ((t\ andalso\ s) \neq FF) \longleftrightarrow t \neq FF \wedge s \neq FF$
 $\langle proof \rangle$

lemma *Def-bool1* [*simp*]: $Def\ x \neq FF \longleftrightarrow x$
 ⟨*proof*⟩

lemma *Def-bool2* [*simp*]: $Def\ x = FF \longleftrightarrow \neg x$
 ⟨*proof*⟩

lemma *Def-bool3* [*simp*]: $Def\ x = TT \longleftrightarrow x$
 ⟨*proof*⟩

lemma *Def-bool4* [*simp*]: $Def\ x \neq TT \longleftrightarrow \neg x$
 ⟨*proof*⟩

lemma *If-and-if*: $(If\ Def\ P\ then\ A\ else\ B) = (if\ P\ then\ A\ else\ B)$
 ⟨*proof*⟩

14.5 Compactness

lemma *compact-TT*: *compact TT*
 ⟨*proof*⟩

lemma *compact-FF*: *compact FF*
 ⟨*proof*⟩

end

15 The type of strict sums

theory *Ssum*
imports *Tr*
begin

default-sort *pcpo*

15.1 Definition of strict sum type

definition *ssum* =
 $\{p :: tr \times ('a \times 'b). p = \perp \vee$
 $(fst\ p = TT \wedge fst\ (snd\ p) \neq \perp \wedge snd\ (snd\ p) = \perp) \vee$
 $(fst\ p = FF \wedge fst\ (snd\ p) = \perp \wedge snd\ (snd\ p) \neq \perp)\}$

pcpodef $('a, 'b)\ ssum\ ((-\oplus/-)\ [21, 20]\ 20) = ssum :: (tr \times 'a \times 'b)\ set$
 ⟨*proof*⟩

instance $ssum :: (\{chfin,pcpo\}, \{chfin,pcpo\})\ chfin$
 ⟨*proof*⟩

type-notation (*ASCII*)
 $ssum\ (infixr\ ++\ 10)$

15.2 Definitions of constructors

definition $\text{sinl} :: 'a \rightarrow ('a ++ 'b)$
where $\text{sinl} = (\Lambda a. \text{Abs-ssum} (\text{seq}\cdot a\cdot \text{TT}, a, \perp))$

definition $\text{sinr} :: 'b \rightarrow ('a ++ 'b)$
where $\text{sinr} = (\Lambda b. \text{Abs-ssum} (\text{seq}\cdot b\cdot \text{FF}, \perp, b))$

lemma $\text{sinl-ssum}: (\text{seq}\cdot a\cdot \text{TT}, a, \perp) \in \text{ssum}$
 $\langle \text{proof} \rangle$

lemma $\text{sinr-ssum}: (\text{seq}\cdot b\cdot \text{FF}, \perp, b) \in \text{ssum}$
 $\langle \text{proof} \rangle$

lemma $\text{Rep-ssum-sinl}: \text{Rep-ssum} (\text{sinl}\cdot a) = (\text{seq}\cdot a\cdot \text{TT}, a, \perp)$
 $\langle \text{proof} \rangle$

lemma $\text{Rep-ssum-sinr}: \text{Rep-ssum} (\text{sinr}\cdot b) = (\text{seq}\cdot b\cdot \text{FF}, \perp, b)$
 $\langle \text{proof} \rangle$

lemmas $\text{Rep-ssum-simps} =$
 Rep-ssum-inject $[\text{symmetric}]$ below-ssum-def
 prod-eq-iff below-prod-def
 Rep-ssum-strict Rep-ssum-sinl Rep-ssum-sinr

15.3 Properties of sinl and sinr

Ordering

lemma $\text{sinl-below} [\text{simp}]: \text{sinl}\cdot x \sqsubseteq \text{sinl}\cdot y \longleftrightarrow x \sqsubseteq y$
 $\langle \text{proof} \rangle$

lemma $\text{sinr-below} [\text{simp}]: \text{sinr}\cdot x \sqsubseteq \text{sinr}\cdot y \longleftrightarrow x \sqsubseteq y$
 $\langle \text{proof} \rangle$

lemma $\text{sinl-below-sinr} [\text{simp}]: \text{sinl}\cdot x \sqsubseteq \text{sinr}\cdot y \longleftrightarrow x = \perp$
 $\langle \text{proof} \rangle$

lemma $\text{sinr-below-sinl} [\text{simp}]: \text{sinr}\cdot x \sqsubseteq \text{sinl}\cdot y \longleftrightarrow x = \perp$
 $\langle \text{proof} \rangle$

Equality

lemma $\text{sinl-eq} [\text{simp}]: \text{sinl}\cdot x = \text{sinl}\cdot y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma $\text{sinr-eq} [\text{simp}]: \text{sinr}\cdot x = \text{sinr}\cdot y \longleftrightarrow x = y$
 $\langle \text{proof} \rangle$

lemma $\text{sinl-eq-sinr} [\text{simp}]: \text{sinl}\cdot x = \text{sinr}\cdot y \longleftrightarrow x = \perp \wedge y = \perp$
 $\langle \text{proof} \rangle$

lemma *sinr-eq-sinl* [*simp*]: $\text{sinr}\cdot x = \text{sinl}\cdot y \longleftrightarrow x = \perp \wedge y = \perp$
 ⟨*proof*⟩

lemma *sinl-inject*: $\text{sinl}\cdot x = \text{sinl}\cdot y \implies x = y$
 ⟨*proof*⟩

lemma *sinr-inject*: $\text{sinr}\cdot x = \text{sinr}\cdot y \implies x = y$
 ⟨*proof*⟩

Strictness

lemma *sinl-strict* [*simp*]: $\text{sinl}\cdot \perp = \perp$
 ⟨*proof*⟩

lemma *sinr-strict* [*simp*]: $\text{sinr}\cdot \perp = \perp$
 ⟨*proof*⟩

lemma *sinl-bottom-iff* [*simp*]: $\text{sinl}\cdot x = \perp \longleftrightarrow x = \perp$
 ⟨*proof*⟩

lemma *sinr-bottom-iff* [*simp*]: $\text{sinr}\cdot x = \perp \longleftrightarrow x = \perp$
 ⟨*proof*⟩

lemma *sinl-defined*: $x \neq \perp \implies \text{sinl}\cdot x \neq \perp$
 ⟨*proof*⟩

lemma *sinr-defined*: $x \neq \perp \implies \text{sinr}\cdot x \neq \perp$
 ⟨*proof*⟩

Compactness

lemma *compact-sinl*: $\text{compact } x \implies \text{compact } (\text{sinl}\cdot x)$
 ⟨*proof*⟩

lemma *compact-sinr*: $\text{compact } x \implies \text{compact } (\text{sinr}\cdot x)$
 ⟨*proof*⟩

lemma *compact-sinlD*: $\text{compact } (\text{sinl}\cdot x) \implies \text{compact } x$
 ⟨*proof*⟩

lemma *compact-sinrD*: $\text{compact } (\text{sinr}\cdot x) \implies \text{compact } x$
 ⟨*proof*⟩

lemma *compact-sinl-iff* [*simp*]: $\text{compact } (\text{sinl}\cdot x) = \text{compact } x$
 ⟨*proof*⟩

lemma *compact-sinr-iff* [*simp*]: $\text{compact } (\text{sinr}\cdot x) = \text{compact } x$
 ⟨*proof*⟩

15.4 Case analysis

lemma *ssumE* [case-names bottom *sinl sinr*, cases type: *ssum*]:

obtains $p = \perp$
 | x **where** $p = \text{sinl}\cdot x$ **and** $x \neq \perp$
 | y **where** $p = \text{sinr}\cdot y$ **and** $y \neq \perp$
 ⟨proof⟩

lemma *ssum-induct* [case-names bottom *sinl sinr*, induct type: *ssum*]:

$\llbracket P \perp;$
 $\bigwedge x. x \neq \perp \implies P (\text{sinl}\cdot x);$
 $\bigwedge y. y \neq \perp \implies P (\text{sinr}\cdot y) \rrbracket \implies P x$
 ⟨proof⟩

lemma *ssumE2* [case-names *sinl sinr*]:

$\llbracket \bigwedge x. p = \text{sinl}\cdot x \implies Q; \bigwedge y. p = \text{sinr}\cdot y \implies Q \rrbracket \implies Q$
 ⟨proof⟩

lemma *below-sinlD*: $p \sqsubseteq \text{sinl}\cdot x \implies \exists y. p = \text{sinl}\cdot y \wedge y \sqsubseteq x$

⟨proof⟩

lemma *below-sinrD*: $p \sqsubseteq \text{sinr}\cdot x \implies \exists y. p = \text{sinr}\cdot y \wedge y \sqsubseteq x$

⟨proof⟩

15.5 Case analysis combinator

definition *sscase* :: $('a \rightarrow 'c) \rightarrow ('b \rightarrow 'c) \rightarrow ('a ++ 'b) \rightarrow 'c$

where *sscase* = $(\Lambda f g s. (\lambda(t, x, y). \text{If } t \text{ then } f\cdot x \text{ else } g\cdot y)) (\text{Rep-ssum } s)$

translations

case *s* of *XCONST sinl*· $x \Rightarrow t1$ | *XCONST sinr*· $y \Rightarrow t2 \Leftrightarrow \text{CONST } \text{sscase}\cdot(\Lambda x.$

$t1)\cdot(\Lambda y. t2)\cdot s$

case *s* of (*XCONST sinl* :: $'a$)· $x \Rightarrow t1$ | *XCONST sinr*· $y \Rightarrow t2 \rightarrow \text{CONST } \text{sscase}\cdot(\Lambda x. t1)\cdot(\Lambda y. t2)\cdot s$

translations

$\Lambda(\text{XCONST } \text{sinl}\cdot x). t \Leftrightarrow \text{CONST } \text{sscase}\cdot(\Lambda x. t)\cdot \perp$

$\Lambda(\text{XCONST } \text{sinr}\cdot y). t \Leftrightarrow \text{CONST } \text{sscase}\cdot \perp\cdot(\Lambda y. t)$

lemma *beta-sscase*: $\text{sscase}\cdot f\cdot g\cdot s = (\lambda(t, x, y). \text{If } t \text{ then } f\cdot x \text{ else } g\cdot y) (\text{Rep-ssum } s)$

⟨proof⟩

lemma *sscase1* [*simp*]: $\text{sscase}\cdot f\cdot g\cdot \perp = \perp$

⟨proof⟩

lemma *sscase2* [*simp*]: $x \neq \perp \implies \text{sscase}\cdot f\cdot g\cdot(\text{sinl}\cdot x) = f\cdot x$

⟨proof⟩

lemma *sscase3* [*simp*]: $y \neq \perp \implies \text{sscase}\cdot f\cdot g\cdot(\text{sinr}\cdot y) = g\cdot y$

⟨proof⟩

lemma *sscase4* [*simp*]: *sscase.sinl.sinr.z = z*
 ⟨*proof*⟩

15.6 Strict sum preserves flatness

instance *ssum* :: (*flat*, *flat*) *flat*
 ⟨*proof*⟩

end

16 The Strict Function Type

theory *Sfun*
imports *Cfun*
begin

pcpodef (*'a*, *'b*) *sfun* (**infixr** $\rightarrow!$ 0) = {*f* :: *'a* \rightarrow *'b*. *f*· \perp = \perp }
 ⟨*proof*⟩

type-notation (*ASCII*)
sfun (**infixr** $\rightarrow!$ 0)

TODO: Define nice syntax for abstraction, application.

definition *sfun-abs* :: (*'a* \rightarrow *'b*) \rightarrow (*'a* $\rightarrow!$ *'b*)
where *sfun-abs* = (Λ *f*. *Abs-sfun* (*strictify*·*f*))

definition *sfun-rep* :: (*'a* $\rightarrow!$ *'b*) \rightarrow *'a* \rightarrow *'b*
where *sfun-rep* = (Λ *f*. *Rep-sfun* *f*)

lemma *sfun-rep-beta*: *sfun-rep*·*f* = *Rep-sfun* *f*
 ⟨*proof*⟩

lemma *sfun-rep-strict1* [*simp*]: *sfun-rep*· \perp = \perp
 ⟨*proof*⟩

lemma *sfun-rep-strict2* [*simp*]: *sfun-rep*·*f*· \perp = \perp
 ⟨*proof*⟩

lemma *strictify-cancel*: *f*· \perp = \perp \implies *strictify*·*f* = *f*
 ⟨*proof*⟩

lemma *sfun-abs-sfun-rep* [*simp*]: *sfun-abs*·(*sfun-rep*·*f*) = *f*
 ⟨*proof*⟩

lemma *sfun-rep-sfun-abs* [*simp*]: *sfun-rep*·(*sfun-abs*·*f*) = *strictify*·*f*
 ⟨*proof*⟩

lemma *sfun-eq-iff*: *f* = *g* \iff *sfun-rep*·*f* = *sfun-rep*·*g*

<proof>

lemma *sfun-below-iff*: $f \sqsubseteq g \iff \text{sfun-rep}\cdot f \sqsubseteq \text{sfun-rep}\cdot g$
<proof>

end

17 Map functions for various types

theory *Map-Functions*

imports *Deflation Sprod Ssum Sfun Up*

begin

17.1 Map operator for continuous function space

default-sort *cpo*

definition *cfun-map* :: $('b \rightarrow 'a) \rightarrow ('c \rightarrow 'd) \rightarrow ('a \rightarrow 'c) \rightarrow ('b \rightarrow 'd)$
where $\text{cfun-map} = (\Lambda a b f x. b \cdot (f \cdot (a \cdot x)))$

lemma *cfun-map-beta* [*simp*]: $\text{cfun-map}\cdot a \cdot b \cdot f \cdot x = b \cdot (f \cdot (a \cdot x))$
<proof>

lemma *cfun-map-ID*: $\text{cfun-map}\cdot \text{ID} \cdot \text{ID} = \text{ID}$
<proof>

lemma *cfun-map-map*: $\text{cfun-map}\cdot f1 \cdot g1 \cdot (\text{cfun-map}\cdot f2 \cdot g2 \cdot p) = \text{cfun-map}\cdot (\Lambda x. f2 \cdot (f1 \cdot x)) \cdot (\Lambda x. g1 \cdot (g2 \cdot x)) \cdot p$
<proof>

lemma *ep-pair-cfun-map*:
assumes *ep-pair e1 p1 and ep-pair e2 p2*
shows *ep-pair (cfun-map\cdot p1 \cdot e2) (cfun-map\cdot e1 \cdot p2)*
<proof>

lemma *deflation-cfun-map*:
assumes *deflation d1 and deflation d2*
shows *deflation (cfun-map\cdot d1 \cdot d2)*
<proof>

lemma *finite-range-cfun-map*:
assumes *a: finite (range (\lambda x. a \cdot x))*
assumes *b: finite (range (\lambda y. b \cdot y))*
shows *finite (range (\lambda f. cfun-map\cdot a \cdot b \cdot f)) (is finite (range ?h))*
<proof>

lemma *finite-deflation-cfun-map*:
assumes *finite-deflation d1 and finite-deflation d2*
shows *finite-deflation (cfun-map\cdot d1 \cdot d2)*

<proof>

Finite deflations are compact elements of the function space

lemma *finite-deflation-imp-compact*: *finite-deflation* $d \implies$ *compact* d
<proof>

17.2 Map operator for product type

definition *prod-map* :: $('a \rightarrow 'b) \rightarrow ('c \rightarrow 'd) \rightarrow 'a \times 'c \rightarrow 'b \times 'd$
where *prod-map* = $(\Lambda f g p. (f \cdot (\text{fst } p), g \cdot (\text{snd } p)))$

lemma *prod-map-Pair* [*simp*]: *prod-map* $\cdot f \cdot g \cdot (x, y) = (f \cdot x, g \cdot y)$
<proof>

lemma *prod-map-ID*: *prod-map* $\cdot \text{ID} \cdot \text{ID} = \text{ID}$
<proof>

lemma *prod-map-map*: *prod-map* $\cdot f1 \cdot g1 \cdot (\text{prod-map} \cdot f2 \cdot g2 \cdot p) = \text{prod-map} \cdot (\Lambda x. f1 \cdot (f2 \cdot x)) \cdot (\Lambda x. g1 \cdot (g2 \cdot x)) \cdot p$
<proof>

lemma *ep-pair-prod-map*:
assumes *ep-pair* $e1$ $p1$ **and** *ep-pair* $e2$ $p2$
shows *ep-pair* $(\text{prod-map} \cdot e1 \cdot e2)$ $(\text{prod-map} \cdot p1 \cdot p2)$
<proof>

lemma *deflation-prod-map*:
assumes *deflation* $d1$ **and** *deflation* $d2$
shows *deflation* $(\text{prod-map} \cdot d1 \cdot d2)$
<proof>

lemma *finite-deflation-prod-map*:
assumes *finite-deflation* $d1$ **and** *finite-deflation* $d2$
shows *finite-deflation* $(\text{prod-map} \cdot d1 \cdot d2)$
<proof>

17.3 Map function for lifted cpo

definition *u-map* :: $('a \rightarrow 'b) \rightarrow 'a \text{ u} \rightarrow 'b \text{ u}$
where *u-map* = $(\Lambda f. \text{fup} \cdot (\text{up } \circ \circ f))$

lemma *u-map-strict* [*simp*]: *u-map* $\cdot f \cdot \perp = \perp$
<proof>

lemma *u-map-up* [*simp*]: *u-map* $\cdot f \cdot (\text{up} \cdot x) = \text{up} \cdot (f \cdot x)$
<proof>

lemma *u-map-ID*: *u-map* $\cdot \text{ID} = \text{ID}$
<proof>

lemma *u-map-map*: $u\text{-map}\cdot f\cdot(u\text{-map}\cdot g\cdot p) = u\text{-map}\cdot(\Lambda x. f\cdot(g\cdot x))\cdot p$
 ⟨proof⟩

lemma *u-map-oo*: $u\text{-map}\cdot(f\text{ oo } g) = u\text{-map}\cdot f\text{ oo } u\text{-map}\cdot g$
 ⟨proof⟩

lemma *ep-pair-u-map*: $ep\text{-pair } e\ p \implies ep\text{-pair } (u\text{-map}\cdot e)\ (u\text{-map}\cdot p)$
 ⟨proof⟩

lemma *deflation-u-map*: $deflation\ d \implies deflation\ (u\text{-map}\cdot d)$
 ⟨proof⟩

lemma *finite-deflation-u-map*:
 assumes *finite-deflation* d
 shows *finite-deflation* $(u\text{-map}\cdot d)$
 ⟨proof⟩

17.4 Map function for strict products

default-sort *pcpo*

definition *sprod-map* :: $('a \rightarrow 'b) \rightarrow ('c \rightarrow 'd) \rightarrow 'a \otimes 'c \rightarrow 'b \otimes 'd$
 where $sprod\text{-map} = (\Lambda f\ g. ssplit\cdot(\Lambda x\ y. (:f\cdot x, g\cdot y)))$

lemma *sprod-map-strict* [*simp*]: $sprod\text{-map}\cdot a\cdot b\cdot \perp = \perp$
 ⟨proof⟩

lemma *sprod-map-spair* [*simp*]: $x \neq \perp \implies y \neq \perp \implies sprod\text{-map}\cdot f\cdot g\cdot (:x, y) = (:f\cdot x, g\cdot y)$
 ⟨proof⟩

lemma *sprod-map-spair'*: $f\cdot \perp = \perp \implies g\cdot \perp = \perp \implies sprod\text{-map}\cdot f\cdot g\cdot (:x, y) = (:f\cdot x, g\cdot y)$
 ⟨proof⟩

lemma *sprod-map-ID*: $sprod\text{-map}\cdot ID\cdot ID = ID$
 ⟨proof⟩

lemma *sprod-map-map*:
 $\llbracket f1\cdot \perp = \perp; g1\cdot \perp = \perp \rrbracket \implies$
 $sprod\text{-map}\cdot f1\cdot g1\cdot (sprod\text{-map}\cdot f2\cdot g2\cdot p) =$
 $sprod\text{-map}\cdot(\Lambda x. f1\cdot(f2\cdot x))\cdot(\Lambda x. g1\cdot(g2\cdot x))\cdot p$
 ⟨proof⟩

lemma *ep-pair-sprod-map*:
 assumes *ep-pair* $e1\ p1$ and *ep-pair* $e2\ p2$
 shows *ep-pair* $(sprod\text{-map}\cdot e1\cdot e2)\ (sprod\text{-map}\cdot p1\cdot p2)$
 ⟨proof⟩

lemma *deflation-sprod-map*:
assumes *deflation d1 and deflation d2*
shows *deflation (sprod-map·d1·d2)*
 ⟨*proof*⟩

lemma *finite-deflation-sprod-map*:
assumes *finite-deflation d1 and finite-deflation d2*
shows *finite-deflation (sprod-map·d1·d2)*
 ⟨*proof*⟩

17.5 Map function for strict sums

definition *ssum-map* :: ($'a \rightarrow 'b$) \rightarrow ($'c \rightarrow 'd$) \rightarrow $'a \oplus 'c \rightarrow 'b \oplus 'd$
where *ssum-map* = $(\Lambda f g. \text{sscase} \cdot (\text{sinl} \text{ oo } f) \cdot (\text{sinr} \text{ oo } g))$

lemma *ssum-map-strict [simp]*: *ssum-map·f·g· \perp = \perp*
 ⟨*proof*⟩

lemma *ssum-map-sinl [simp]*: $x \neq \perp \implies \text{ssum-map} \cdot f \cdot g \cdot (\text{sinl} \cdot x) = \text{sinl} \cdot (f \cdot x)$
 ⟨*proof*⟩

lemma *ssum-map-sinr [simp]*: $x \neq \perp \implies \text{ssum-map} \cdot f \cdot g \cdot (\text{sinr} \cdot x) = \text{sinr} \cdot (g \cdot x)$
 ⟨*proof*⟩

lemma *ssum-map-sinl'*: $f \cdot \perp = \perp \implies \text{ssum-map} \cdot f \cdot g \cdot (\text{sinl} \cdot x) = \text{sinl} \cdot (f \cdot x)$
 ⟨*proof*⟩

lemma *ssum-map-sinr'*: $g \cdot \perp = \perp \implies \text{ssum-map} \cdot f \cdot g \cdot (\text{sinr} \cdot x) = \text{sinr} \cdot (g \cdot x)$
 ⟨*proof*⟩

lemma *ssum-map-ID*: *ssum-map·ID·ID = ID*
 ⟨*proof*⟩

lemma *ssum-map-map*:
 $\llbracket f1 \cdot \perp = \perp; g1 \cdot \perp = \perp \rrbracket \implies$
 $\text{ssum-map} \cdot f1 \cdot g1 \cdot (\text{ssum-map} \cdot f2 \cdot g2 \cdot p) =$
 $\text{ssum-map} \cdot (\Lambda x. f1 \cdot (f2 \cdot x)) \cdot (\Lambda x. g1 \cdot (g2 \cdot x)) \cdot p$
 ⟨*proof*⟩

lemma *ep-pair-ssum-map*:
assumes *ep-pair e1 p1 and ep-pair e2 p2*
shows *ep-pair (ssum-map·e1·e2) (ssum-map·p1·p2)*
 ⟨*proof*⟩

lemma *deflation-ssum-map*:
assumes *deflation d1 and deflation d2*
shows *deflation (ssum-map·d1·d2)*
 ⟨*proof*⟩

lemma *finite-deflation-ssum-map*:
 assumes *finite-deflation d1* and *finite-deflation d2*
 shows *finite-deflation (ssum-map.d1.d2)*
 ⟨*proof*⟩

17.6 Map operator for strict function space

definition *sfun-map* :: $('b \rightarrow 'a) \rightarrow ('c \rightarrow 'd) \rightarrow ('a \rightarrow! 'c) \rightarrow ('b \rightarrow! 'd)$
 where *sfun-map* = $(\Lambda a b. \text{sfun-abs } oo \text{ cfun-map.a.b } oo \text{ sfun-rep})$

lemma *sfun-map-ID*: *sfun-map.ID.ID = ID*
 ⟨*proof*⟩

lemma *sfun-map-map*:
 assumes $f2.\perp = \perp$ and $g2.\perp = \perp$
 shows $\text{sfun-map.f1.g1} \cdot (\text{sfun-map.f2.g2.p}) =$
 $\text{sfun-map} \cdot (\Lambda x. f2 \cdot (f1 \cdot x)) \cdot (\Lambda x. g1 \cdot (g2 \cdot x)) \cdot p$
 ⟨*proof*⟩

lemma *ep-pair-sfun-map*:
 assumes 1: *ep-pair e1 p1*
 assumes 2: *ep-pair e2 p2*
 shows *ep-pair (sfun-map.p1.e2) (sfun-map.e1.p2)*
 ⟨*proof*⟩

lemma *deflation-sfun-map*:
 assumes 1: *deflation d1*
 assumes 2: *deflation d2*
 shows *deflation (sfun-map.d1.d2)*
 ⟨*proof*⟩

lemma *finite-deflation-sfun-map*:
 assumes *finite-deflation d1*
 and *finite-deflation d2*
 shows *finite-deflation (sfun-map.d1.d2)*
 ⟨*proof*⟩

end

18 The cpo of cartesian products

theory *Cprod*
 imports *Cfun*
 begin

 default-sort *cpo*

18.1 Continuous case function for unit type

definition *unit-when* :: 'a → unit → 'a
 where *unit-when* = (λ a -. a)

translations

Λ(). *t* ⇐ CONST *unit-when*·*t*

lemma *unit-when* [simp]: *unit-when*·a·u = a
 ⟨proof⟩

18.2 Continuous version of split function

definition *csplit* :: ('a → 'b → 'c) → ('a × 'b) → 'c
 where *csplit* = (λ f p. f·(fst p)·(snd p))

translations

Λ(CONST Pair *x y*). *t* ⇐ CONST *csplit*·(Λ *x y*· *t*)

abbreviation *cfst* :: 'a × 'b → 'a
 where *cfst* ≡ Abs-cfun *fst*

abbreviation *csnd* :: 'a × 'b → 'b
 where *csnd* ≡ Abs-cfun *snd*

18.3 Convert all lemmas to the continuous versions

lemma *csplit1* [simp]: *csplit*·f·⊥ = f·⊥·⊥
 ⟨proof⟩

lemma *csplit-Pair* [simp]: *csplit*·f·(x, y) = f·x·y
 ⟨proof⟩

end

19 Profinite and bifinite cpos

theory *Bifinite*

imports *Map-Functions Cprod Sprod Sfun Up HOL-Library.Countable*
 begin

default-sort *cpo*

19.1 Chains of finite deflations

locale *approx-chain* =

fixes *approx* :: nat ⇒ 'a → 'a

assumes *chain-approx* [simp]: *chain* (λi. *approx* i)

assumes *lub-approx* [simp]: (⊔ i. *approx* i) = ID

assumes *finite-deflation-approx* [simp]: ∧i. *finite-deflation* (*approx* i)

begin

lemma *deflation-approx*: *deflation* (*approx i*)
 ⟨*proof*⟩

lemma *approx-idem*: *approx i*·(*approx i*·*x*) = *approx i*·*x*
 ⟨*proof*⟩

lemma *approx-below*: *approx i*·*x* \sqsubseteq *x*
 ⟨*proof*⟩

lemma *finite-range-approx*: *finite* (*range* ($\lambda x. \text{approx } i \cdot x$))
 ⟨*proof*⟩

lemma *compact-approx [simp]*: *compact* (*approx n*·*x*)
 ⟨*proof*⟩

lemma *compact-eq-approx*: *compact x* $\implies \exists i. \text{approx } i \cdot x = x$
 ⟨*proof*⟩

end

19.2 Omega-profinite and bifinite domains

class *bifinite* = *pcpo* +
assumes *bifinite*: $\exists (a::\text{nat} \Rightarrow 'a \rightarrow 'a). \text{approx-chain } a$

class *profinite* = *cpo* +
assumes *profinite*: $\exists (a::\text{nat} \Rightarrow 'a_{\perp} \rightarrow 'a_{\perp}). \text{approx-chain } a$

19.3 Building approx chains

lemma *approx-chain-iso*:
assumes *a*: *approx-chain a*
assumes [*simp*]: $\bigwedge x. f \cdot (g \cdot x) = x$
assumes [*simp*]: $\bigwedge y. g \cdot (f \cdot y) = y$
shows *approx-chain* ($\lambda i. f \text{ oo } a \text{ i oo } g$)
 ⟨*proof*⟩

lemma *approx-chain-u-map*:
assumes *approx-chain a*
shows *approx-chain* ($\lambda i. \text{u-map} \cdot (a \text{ i})$)
 ⟨*proof*⟩

lemma *approx-chain-sfun-map*:
assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* ($\lambda i. \text{sfun-map} \cdot (a \text{ i}) \cdot (b \text{ i})$)
 ⟨*proof*⟩

lemma *approx-chain-sprod-map*:

assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* $(\lambda i. \text{sprod-map} \cdot (a \ i) \cdot (b \ i))$
 $\langle \text{proof} \rangle$

lemma *approx-chain-ssum-map*:
assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* $(\lambda i. \text{ssum-map} \cdot (a \ i) \cdot (b \ i))$
 $\langle \text{proof} \rangle$

lemma *approx-chain-cfun-map*:
assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* $(\lambda i. \text{cfun-map} \cdot (a \ i) \cdot (b \ i))$
 $\langle \text{proof} \rangle$

lemma *approx-chain-prod-map*:
assumes *approx-chain a* **and** *approx-chain b*
shows *approx-chain* $(\lambda i. \text{prod-map} \cdot (a \ i) \cdot (b \ i))$
 $\langle \text{proof} \rangle$

Approx chains for countable discrete types.

definition *discr-approx* :: $\text{nat} \Rightarrow 'a::\text{countable} \text{discr } u \rightarrow 'a \text{discr } u$
where *discr-approx* = $(\lambda i. \Lambda(\text{up} \cdot x). \text{if to-nat } (\text{undiscr } x) < i \text{ then } \text{up} \cdot x \text{ else } \perp)$

lemma *chain-discr-approx* [*simp*]: *chain* *discr-approx*
 $\langle \text{proof} \rangle$

lemma *lub-discr-approx* [*simp*]: $(\bigsqcup i. \text{discr-approx } i) = \text{ID}$
 $\langle \text{proof} \rangle$

lemma *inj-on-undiscr* [*simp*]: *inj-on* *undiscr A*
 $\langle \text{proof} \rangle$

lemma *finite-deflation-discr-approx*: *finite-deflation* (*discr-approx i*)
 $\langle \text{proof} \rangle$

lemma *discr-approx*: *approx-chain* *discr-approx*
 $\langle \text{proof} \rangle$

19.4 Class instance proofs

instance *bifinite* \subseteq *profinite*
 $\langle \text{proof} \rangle$

instance $u :: (\text{profinite}) \text{bifinite}$
 $\langle \text{proof} \rangle$

Types $'a \rightarrow 'b$ and $'a_{\perp} \rightarrow! 'b$ are isomorphic.

definition *encode-cfun* = $(\Lambda f. \text{sfun-abs} \cdot (\text{fup} \cdot f))$

definition $decode-cfun = (\Lambda g x. sfun-rep.g \cdot (up \cdot x))$

lemma $decode-encode-cfun$ [simp]: $decode-cfun \cdot (encode-cfun \cdot x) = x$
 $\langle proof \rangle$

lemma $encode-decode-cfun$ [simp]: $encode-cfun \cdot (decode-cfun \cdot y) = y$
 $\langle proof \rangle$

instance $cfun :: (profinite, bifinite) bifinite$
 $\langle proof \rangle$

Types $(\prime a \times \prime b)_\perp$ and $\prime a_\perp \otimes \prime b_\perp$ are isomorphic.

definition $encode-prod-u = (\Lambda (up \cdot (x, y)). (:up \cdot x, up \cdot y))$

definition $decode-prod-u = (\Lambda (:up \cdot x, up \cdot y). up \cdot (x, y))$

lemma $decode-encode-prod-u$ [simp]: $decode-prod-u \cdot (encode-prod-u \cdot x) = x$
 $\langle proof \rangle$

lemma $encode-decode-prod-u$ [simp]: $encode-prod-u \cdot (decode-prod-u \cdot y) = y$
 $\langle proof \rangle$

instance $prod :: (profinite, profinite) profinite$
 $\langle proof \rangle$

instance $prod :: (bifinite, bifinite) bifinite$
 $\langle proof \rangle$

instance $sfun :: (bifinite, bifinite) bifinite$
 $\langle proof \rangle$

instance $sprod :: (bifinite, bifinite) bifinite$
 $\langle proof \rangle$

instance $ssum :: (bifinite, bifinite) bifinite$
 $\langle proof \rangle$

lemma $approx-chain-unit$: $approx-chain (\perp :: nat \Rightarrow unit \rightarrow unit)$
 $\langle proof \rangle$

instance $unit :: bifinite$
 $\langle proof \rangle$

instance $discr :: (countable) profinite$
 $\langle proof \rangle$

instance $lift :: (countable) bifinite$
 $\langle proof \rangle$

end

20 Defining algebraic domains by ideal completion

```
theory Completion
imports Cfun
begin
```

20.1 Ideals over a preorder

```
locale preorder =
  fixes r :: 'a::type  $\Rightarrow$  'a  $\Rightarrow$  bool (infix  $\preceq$  50)
  assumes r-refl:  $x \preceq x$ 
  assumes r-trans:  $\llbracket x \preceq y; y \preceq z \rrbracket \Longrightarrow x \preceq z$ 
begin
```

definition

```
ideal :: 'a set  $\Rightarrow$  bool where
ideal A = (( $\exists x. x \in A$ )  $\wedge$  ( $\forall x \in A. \forall y \in A. \exists z \in A. x \preceq z \wedge y \preceq z$ )  $\wedge$ 
  ( $\forall x y. x \preceq y \longrightarrow y \in A \longrightarrow x \in A$ ))
```

lemma idealI:

```
assumes  $\exists x. x \in A$ 
assumes  $\bigwedge x y. \llbracket x \in A; y \in A \rrbracket \Longrightarrow \exists z \in A. x \preceq z \wedge y \preceq z$ 
assumes  $\bigwedge x y. \llbracket x \preceq y; y \in A \rrbracket \Longrightarrow x \in A$ 
shows ideal A
```

<proof>

lemma idealD1:

```
ideal A  $\Longrightarrow \exists x. x \in A$ 
```

<proof>

lemma idealD2:

```
 $\llbracket \text{ideal } A; x \in A; y \in A \rrbracket \Longrightarrow \exists z \in A. x \preceq z \wedge y \preceq z$ 
```

<proof>

lemma idealD3:

```
 $\llbracket \text{ideal } A; x \preceq y; y \in A \rrbracket \Longrightarrow x \in A$ 
```

<proof>

lemma ideal-principal: ideal $\{x. x \preceq z\}$

<proof>

lemma ex-ideal: $\exists A. A \in \{A. \text{ideal } A\}$

<proof>

The set of ideals is a cpo

lemma ideal-UN:

```
fixes A :: nat  $\Rightarrow$  'a set
```

assumes *ideal-A*: $\bigwedge i. \text{ideal } (A \ i)$
assumes *chain-A*: $\bigwedge i \ j. i \leq j \implies A \ i \subseteq A \ j$
shows *ideal* $(\bigcup i. A \ i)$
 <proof>

lemma *typedef-ideal-po*:
fixes *Abs* :: 'a set \Rightarrow 'b::below
assumes *type*: *type-definition* *Rep* *Abs* {*S*. *ideal* *S*}
assumes *below*: $\bigwedge x \ y. x \sqsubseteq y \longleftrightarrow \text{Rep } x \subseteq \text{Rep } y$
shows *OFCLASS*('b, *po-class*)
 <proof>

lemma
fixes *Abs* :: 'a set \Rightarrow 'b::po
assumes *type*: *type-definition* *Rep* *Abs* {*S*. *ideal* *S*}
assumes *below*: $\bigwedge x \ y. x \sqsubseteq y \longleftrightarrow \text{Rep } x \subseteq \text{Rep } y$
assumes *S*: *chain* *S*
shows *typedef-ideal-lub*: $\text{range } S \ll\mid \text{Abs } (\bigcup i. \text{Rep } (S \ i))$
and *typedef-ideal-rep-lub*: $\text{Rep } (\bigsqcup i. S \ i) = (\bigcup i. \text{Rep } (S \ i))$
 <proof>

lemma *typedef-ideal-cpo*:
fixes *Abs* :: 'a set \Rightarrow 'b::po
assumes *type*: *type-definition* *Rep* *Abs* {*S*. *ideal* *S*}
assumes *below*: $\bigwedge x \ y. x \sqsubseteq y \longleftrightarrow \text{Rep } x \subseteq \text{Rep } y$
shows *OFCLASS*('b, *cpo-class*)
 <proof>

end

interpretation *below*: *preorder* *below* :: 'a::po \Rightarrow 'a \Rightarrow *bool*
 <proof>

20.2 Lemmas about least upper bounds

lemma *is-ub-the-lub-ex*: $[\exists u. S \ll\mid u; x \in S] \implies x \sqsubseteq \text{lub } S$
 <proof>

lemma *is-lub-the-lub-ex*: $[\exists u. S \ll\mid u; S <\mid x] \implies \text{lub } S \sqsubseteq x$
 <proof>

20.3 Locale for ideal completion

hide-const (open) *Filter.principal*

locale *ideal-completion* = *preorder* +
fixes *principal* :: 'a::type \Rightarrow 'b::cpo
fixes *rep* :: 'b::cpo \Rightarrow 'a::type set
assumes *ideal-rep*: $\bigwedge x. \text{ideal } (\text{rep } x)$
assumes *rep-lub*: $\bigwedge Y. \text{chain } Y \implies \text{rep } (\bigsqcup i. Y \ i) = (\bigcup i. \text{rep } (Y \ i))$

assumes *rep-principal*: $\bigwedge a. \text{rep } (\text{principal } a) = \{b. b \preceq a\}$
assumes *belowI*: $\bigwedge x y. \text{rep } x \subseteq \text{rep } y \implies x \sqsubseteq y$
assumes *countable*: $\exists f::'a \Rightarrow \text{nat. inj } f$

begin

lemma *rep-mono*: $x \sqsubseteq y \implies \text{rep } x \subseteq \text{rep } y$
 $\langle \text{proof} \rangle$

lemma *below-def*: $x \sqsubseteq y \longleftrightarrow \text{rep } x \subseteq \text{rep } y$
 $\langle \text{proof} \rangle$

lemma *principal-below-iff-mem-rep*: $\text{principal } a \sqsubseteq x \longleftrightarrow a \in \text{rep } x$
 $\langle \text{proof} \rangle$

lemma *principal-below-iff [simp]*: $\text{principal } a \sqsubseteq \text{principal } b \longleftrightarrow a \preceq b$
 $\langle \text{proof} \rangle$

lemma *principal-eq-iff*: $\text{principal } a = \text{principal } b \longleftrightarrow a \preceq b \wedge b \preceq a$
 $\langle \text{proof} \rangle$

lemma *eq-iff*: $x = y \longleftrightarrow \text{rep } x = \text{rep } y$
 $\langle \text{proof} \rangle$

lemma *principal-mono*: $a \preceq b \implies \text{principal } a \sqsubseteq \text{principal } b$
 $\langle \text{proof} \rangle$

lemma *ch2ch-principal [simp]*:
 $\forall i. Y i \preceq Y (\text{Suc } i) \implies \text{chain } (\lambda i. \text{principal } (Y i))$
 $\langle \text{proof} \rangle$

20.3.1 Principal ideals approximate all elements

lemma *compact-principal [simp]*: $\text{compact } (\text{principal } a)$
 $\langle \text{proof} \rangle$

Construct a chain whose lub is the same as a given ideal

lemma *obtain-principal-chain*:
obtains Y **where** $\forall i. Y i \preceq Y (\text{Suc } i)$ **and** $x = (\bigsqcup i. \text{principal } (Y i))$
 $\langle \text{proof} \rangle$

lemma *principal-induct*:
assumes *adm*: $\text{adm } P$
assumes $P: \bigwedge a. P (\text{principal } a)$
shows $P x$
 $\langle \text{proof} \rangle$

lemma *compact-imp-principal*: $\text{compact } x \implies \exists a. x = \text{principal } a$
 $\langle \text{proof} \rangle$

20.4 Defining functions in terms of basis elements

definition

$extension :: ('a::type \Rightarrow 'c::cpo) \Rightarrow 'b \rightarrow 'c$ **where**
 $extension = (\lambda f. (\Lambda x. lub (f \text{ ' rep } x)))$

lemma *extension-lemma*:

fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes $f\text{-mono}: \bigwedge a b. a \preceq b \implies f a \sqsubseteq f b$
shows $\exists u. f \text{ ' rep } x \ll\mid u$

<proof>

lemma *extension-beta*:

fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes $f\text{-mono}: \bigwedge a b. a \preceq b \implies f a \sqsubseteq f b$
shows $extension f \cdot x = lub (f \text{ ' rep } x)$

<proof>

lemma *extension-principal*:

fixes $f :: 'a::type \Rightarrow 'c::cpo$
assumes $f\text{-mono}: \bigwedge a b. a \preceq b \implies f a \sqsubseteq f b$
shows $extension f \cdot (principal a) = f a$

<proof>

lemma *extension-mono*:

assumes $f\text{-mono}: \bigwedge a b. a \preceq b \implies f a \sqsubseteq f b$
assumes $g\text{-mono}: \bigwedge a b. a \preceq b \implies g a \sqsubseteq g b$
assumes $below: \bigwedge a. f a \sqsubseteq g a$
shows $extension f \sqsubseteq extension g$

<proof>

lemma *cont-extension*:

assumes $f\text{-mono}: \bigwedge a b x. a \preceq b \implies f x a \sqsubseteq f x b$
assumes $f\text{-cont}: \bigwedge a. cont (\lambda x. f x a)$
shows $cont (\lambda x. extension (\lambda a. f x a))$

<proof>

end

lemma (in *preorder*) *typedef-ideal-completion*:

fixes $Abs :: 'a \text{ set} \Rightarrow 'b::cpo$
assumes $type: \text{type-definition } Rep \text{ Abs } \{S. \text{ ideal } S\}$
assumes $below: \bigwedge x y. x \sqsubseteq y \iff Rep x \subseteq Rep y$
assumes $principal: \bigwedge a. principal a = Abs \{b. b \preceq a\}$
assumes $countable: \exists f::'a \Rightarrow nat. inj f$
shows $ideal\text{-completion } r \text{ principal } Rep$

<proof>

end

21 A universal bifinite domain

```
theory Universal
imports Bifinite Completion HOL-Library.Nat-Bijection
begin
```

```
no-notation binomial (infixl choose 65)
```

21.1 Basis for universal domain

21.1.1 Basis datatype

```
type-synonym ubasis = nat
```

definition

```
node :: nat ⇒ ubasis ⇒ ubasis set ⇒ ubasis
```

where

```
node i a S = Suc (prod-encode (i, prod-encode (a, set-encode S)))
```

```
lemma node-not-0 [simp]: node i a S ≠ 0
⟨proof⟩
```

```
lemma node-gt-0 [simp]: 0 < node i a S
⟨proof⟩
```

```
lemma node-inject [simp]:
  [[finite S; finite T]
  ⇒ node i a S = node j b T ↔ i = j ∧ a = b ∧ S = T]
⟨proof⟩
```

```
lemma node-gt0: i < node i a S
⟨proof⟩
```

```
lemma node-gt1: a < node i a S
⟨proof⟩
```

```
lemma nat-less-power2: n < 2n
⟨proof⟩
```

```
lemma node-gt2: [[finite S; b ∈ S] ⇒ b < node i a S]
⟨proof⟩
```

```
lemma eq-prod-encode-pairI:
  [[fst (prod-decode x) = a; snd (prod-decode x) = b] ⇒ x = prod-encode (a, b)]
⟨proof⟩
```

lemma node-cases:

```
assumes 1: x = 0 ⇒ P
```

```
assumes 2: ∧i a S. [[finite S; x = node i a S] ⇒ P
```

```
shows P
```

$\langle proof \rangle$

lemma *node-induct*:

assumes 1: $P\ 0$

assumes 2: $\bigwedge i\ a\ S. \llbracket P\ a; \text{finite } S; \forall b \in S. P\ b \rrbracket \implies P\ (\text{node } i\ a\ S)$

shows $P\ x$

$\langle proof \rangle$

21.1.2 Basis ordering

inductive

ubasis-le :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{bool}$

where

ubasis-le-refl: $\text{ubasis-le } a\ a$

| *ubasis-le-trans*:

$\llbracket \text{ubasis-le } a\ b; \text{ubasis-le } b\ c \rrbracket \implies \text{ubasis-le } a\ c$

| *ubasis-le-lower*:

$\text{finite } S \implies \text{ubasis-le } a\ (\text{node } i\ a\ S)$

| *ubasis-le-upper*:

$\llbracket \text{finite } S; b \in S; \text{ubasis-le } a\ b \rrbracket \implies \text{ubasis-le } (\text{node } i\ a\ S)\ b$

lemma *ubasis-le-minimal*: $\text{ubasis-le } 0\ x$

$\langle proof \rangle$

interpretation *udom*: *preorder* *ubasis-le*

$\langle proof \rangle$

21.1.3 Generic take function

function

ubasis-until :: $(\text{ubasis} \Rightarrow \text{bool}) \Rightarrow \text{ubasis} \Rightarrow \text{ubasis}$

where

ubasis-until $P\ 0 = 0$

| $\text{finite } S \implies \text{ubasis-until } P\ (\text{node } i\ a\ S) =$

(if $P\ (\text{node } i\ a\ S)$ then $\text{node } i\ a\ S$ else *ubasis-until* $P\ a$)

$\langle proof \rangle$

termination *ubasis-until*

$\langle proof \rangle$

lemma *ubasis-until*: $P\ 0 \implies P\ (\text{ubasis-until } P\ x)$

$\langle proof \rangle$

lemma *ubasis-until'*: $0 < \text{ubasis-until } P\ x \implies P\ (\text{ubasis-until } P\ x)$

$\langle proof \rangle$

lemma *ubasis-until-same*: $P\ x \implies \text{ubasis-until } P\ x = x$

$\langle proof \rangle$

lemma *ubasis-until-idem*:

$P\ 0 \implies \text{ubasis-until } P\ (\text{ubasis-until } P\ x) = \text{ubasis-until } P\ x$
 ⟨proof⟩

lemma *ubasis-until-0*:

$\forall x. x \neq 0 \longrightarrow \neg P\ x \implies \text{ubasis-until } P\ x = 0$
 ⟨proof⟩

lemma *ubasis-until-less*: $\text{ubasis-le}\ (\text{ubasis-until } P\ x)\ x$
 ⟨proof⟩

lemma *ubasis-until-chain*:

assumes $PQ: \bigwedge x. P\ x \implies Q\ x$
shows $\text{ubasis-le}\ (\text{ubasis-until } P\ x)\ (\text{ubasis-until } Q\ x)$
 ⟨proof⟩

lemma *ubasis-until-mono*:

assumes $\bigwedge i\ a\ S\ b. \llbracket \text{finite } S; P\ (\text{node } i\ a\ S); b \in S; \text{ubasis-le } a\ b \rrbracket \implies P\ b$
shows $\text{ubasis-le } a\ b \implies \text{ubasis-le}\ (\text{ubasis-until } P\ a)\ (\text{ubasis-until } P\ b)$
 ⟨proof⟩

lemma *finite-range-ubasis-until*:

$\text{finite } \{x. P\ x\} \implies \text{finite}\ (\text{range}\ (\text{ubasis-until } P))$
 ⟨proof⟩

21.2 Defining the universal domain by ideal completion

typedef $\text{udom} = \{S. \text{udom.ideal } S\}$
 ⟨proof⟩

instantiation $\text{udom} :: \text{below}$
begin

definition

$x \sqsubseteq y \longleftrightarrow \text{Rep-udom } x \subseteq \text{Rep-udom } y$

instance ⟨proof⟩
end

instance $\text{udom} :: \text{po}$
 ⟨proof⟩

instance $\text{udom} :: \text{cpo}$
 ⟨proof⟩

definition

$\text{udom-principal} :: \text{nat} \Rightarrow \text{udom}$ **where**
 $\text{udom-principal } t = \text{Abs-udom } \{u. \text{ubasis-le } u\ t\}$

lemma *ubasis-countable*: $\exists f :: \text{ubasis} \Rightarrow \text{nat. inj } f$

<proof>

interpretation *udom*:

ideal-completion ubasis-le udom-principal Rep-udom

<proof>

Universal domain is pointed

lemma *udom-minimal*: *udom-principal* $0 \sqsubseteq x$

<proof>

instance *udom* :: *pcpo*

<proof>

lemma *inst-udom-pcpo*: $\perp = \text{udom-principal } 0$

<proof>

21.3 Compact bases of domains

typedef *'a compact-basis* = $\{x :: 'a :: \text{pcpo. compact } x\}$

<proof>

lemma *Rep-compact-basis'* [*simp*]: *compact* (*Rep-compact-basis* *a*)

<proof>

lemma *Abs-compact-basis-inverse'* [*simp*]:

compact *x* $\implies \text{Rep-compact-basis } (\text{Abs-compact-basis } x) = x$

<proof>

instantiation *compact-basis* :: (*pcpo*) *below*

begin

definition

compact-le-def:

$(\sqsubseteq) \equiv (\lambda x y. \text{Rep-compact-basis } x \sqsubseteq \text{Rep-compact-basis } y)$

instance *<proof>*

end

instance *compact-basis* :: (*pcpo*) *po*

<proof>

definition

approximants :: *'a* \Rightarrow *'a compact-basis set* **where**

approximants = $(\lambda x. \{a. \text{Rep-compact-basis } a \sqsubseteq x\})$

definition

compact-bot :: *'a*::*pcpo compact-basis* **where**

compact-bot = *Abs-compact-basis* \perp

lemma *Rep-compact-bot* [*simp*]: *Rep-compact-basis compact-bot* = \perp
 ⟨*proof*⟩

lemma *compact-bot-minimal* [*simp*]: *compact-bot* \sqsubseteq *a*
 ⟨*proof*⟩

21.4 Universality of *udom*

We use a locale to parameterize the construction over a chain of approx functions on the type to be embedded.

locale *bifinite-approx-chain* =
approx-chain approx for approx :: nat \Rightarrow *'a::bifinite* \rightarrow *'a*
begin

21.4.1 Choosing a maximal element from a finite set

lemma *finite-has-maximal*:
fixes A :: 'a compact-basis set
shows $\llbracket \text{finite } A; A \neq \{\} \rrbracket \Longrightarrow \exists x \in A. \forall y \in A. x \sqsubseteq y \longrightarrow x = y$
 ⟨*proof*⟩

definition
choose :: 'a compact-basis set \Rightarrow *'a compact-basis*
where
choose A = (*SOME x. x* \in $\{x \in A. \forall y \in A. x \sqsubseteq y \longrightarrow x = y\}$)

lemma *choose-lemma*:
 $\llbracket \text{finite } A; A \neq \{\} \rrbracket \Longrightarrow \text{choose } A \in \{x \in A. \forall y \in A. x \sqsubseteq y \longrightarrow x = y\}$
 ⟨*proof*⟩

lemma *maximal-choose*:
 $\llbracket \text{finite } A; y \in A; \text{choose } A \sqsubseteq y \rrbracket \Longrightarrow \text{choose } A = y$
 ⟨*proof*⟩

lemma *choose-in*: $\llbracket \text{finite } A; A \neq \{\} \rrbracket \Longrightarrow \text{choose } A \in A$
 ⟨*proof*⟩

function
choose-pos :: 'a compact-basis set \Rightarrow *'a compact-basis* \Rightarrow *nat*
where
choose-pos A x =
 (*if finite A* \wedge *x* \in *A* \wedge *x* \neq *choose A*
 then *Suc (choose-pos (A - {choose A}) x)* else 0)
 ⟨*proof*⟩

termination *choose-pos*
 ⟨*proof*⟩

declare *choose-pos.simps* [*simp del*]

lemma *choose-pos-choose*: $\text{finite } A \implies \text{choose-pos } A (\text{choose } A) = 0$
 ⟨proof⟩

lemma *inj-on-choose-pos* [OF refl]:
 $\llbracket \text{card } A = n; \text{finite } A \rrbracket \implies \text{inj-on } (\text{choose-pos } A) A$
 ⟨proof⟩

lemma *choose-pos-bounded* [OF refl]:
 $\llbracket \text{card } A = n; \text{finite } A; x \in A \rrbracket \implies \text{choose-pos } A x < n$
 ⟨proof⟩

lemma *choose-pos-lessD*:
 $\llbracket \text{choose-pos } A x < \text{choose-pos } A y; \text{finite } A; x \in A; y \in A \rrbracket \implies x \not\sqsubseteq y$
 ⟨proof⟩

21.4.2 Compact basis take function

primrec

cb-take :: $\text{nat} \Rightarrow 'a \text{ compact-basis} \Rightarrow 'a \text{ compact-basis}$ **where**
cb-take 0 = $(\lambda x. \text{compact-bot})$
 | *cb-take* (Suc n) = $(\lambda a. \text{Abs-compact-basis } (\text{approx } n \cdot (\text{Rep-compact-basis } a)))$

declare *cb-take.simps* [simp del]

lemma *cb-take-zero* [simp]: *cb-take* 0 a = *compact-bot*
 ⟨proof⟩

lemma *Rep-cb-take*:
 $\text{Rep-compact-basis } (\text{cb-take } (\text{Suc } n) a) = \text{approx } n \cdot (\text{Rep-compact-basis } a)$
 ⟨proof⟩

lemmas *approx-Rep-compact-basis* = *Rep-cb-take* [symmetric]

lemma *cb-take-covers*: $\exists n. \text{cb-take } n x = x$
 ⟨proof⟩

lemma *cb-take-less*: $\text{cb-take } n x \sqsubseteq x$
 ⟨proof⟩

lemma *cb-take-idem*: $\text{cb-take } n (\text{cb-take } n x) = \text{cb-take } n x$
 ⟨proof⟩

lemma *cb-take-mono*: $x \sqsubseteq y \implies \text{cb-take } n x \sqsubseteq \text{cb-take } n y$
 ⟨proof⟩

lemma *cb-take-chain-le*: $m \leq n \implies \text{cb-take } m x \sqsubseteq \text{cb-take } n x$
 ⟨proof⟩

lemma *finite-range-cb-take*: $\text{finite } (\text{range } (\text{cb-take } n))$
 ⟨*proof*⟩

21.4.3 Rank of basis elements

definition

$\text{rank} :: 'a \text{ compact-basis} \Rightarrow \text{nat}$

where

$\text{rank } x = (\text{LEAST } n. \text{cb-take } n \ x = x)$

lemma *compact-approx-rank*: $\text{cb-take } (\text{rank } x) \ x = x$
 ⟨*proof*⟩

lemma *rank-leD*: $\text{rank } x \leq n \Longrightarrow \text{cb-take } n \ x = x$
 ⟨*proof*⟩

lemma *rank-leI*: $\text{cb-take } n \ x = x \Longrightarrow \text{rank } x \leq n$
 ⟨*proof*⟩

lemma *rank-le-iff*: $\text{rank } x \leq n \longleftrightarrow \text{cb-take } n \ x = x$
 ⟨*proof*⟩

lemma *rank-compact-bot* [*simp*]: $\text{rank } \text{compact-bot} = 0$
 ⟨*proof*⟩

lemma *rank-eq-0-iff* [*simp*]: $\text{rank } x = 0 \longleftrightarrow x = \text{compact-bot}$
 ⟨*proof*⟩

definition

$\text{rank-le} :: 'a \text{ compact-basis} \Rightarrow 'a \text{ compact-basis set}$

where

$\text{rank-le } x = \{y. \text{rank } y \leq \text{rank } x\}$

definition

$\text{rank-lt} :: 'a \text{ compact-basis} \Rightarrow 'a \text{ compact-basis set}$

where

$\text{rank-lt } x = \{y. \text{rank } y < \text{rank } x\}$

definition

$\text{rank-eq} :: 'a \text{ compact-basis} \Rightarrow 'a \text{ compact-basis set}$

where

$\text{rank-eq } x = \{y. \text{rank } y = \text{rank } x\}$

lemma *rank-eq-cong*: $\text{rank } x = \text{rank } y \Longrightarrow \text{rank-eq } x = \text{rank-eq } y$
 ⟨*proof*⟩

lemma *rank-lt-cong*: $\text{rank } x = \text{rank } y \Longrightarrow \text{rank-lt } x = \text{rank-lt } y$
 ⟨*proof*⟩

lemma *rank-eq-subset*: $\text{rank-eq } x \subseteq \text{rank-le } x$
 ⟨proof⟩

lemma *rank-lt-subset*: $\text{rank-lt } x \subseteq \text{rank-le } x$
 ⟨proof⟩

lemma *finite-rank-le*: $\text{finite } (\text{rank-le } x)$
 ⟨proof⟩

lemma *finite-rank-eq*: $\text{finite } (\text{rank-eq } x)$
 ⟨proof⟩

lemma *finite-rank-lt*: $\text{finite } (\text{rank-lt } x)$
 ⟨proof⟩

lemma *rank-lt-Int-rank-eq*: $\text{rank-lt } x \cap \text{rank-eq } x = \{\}$
 ⟨proof⟩

lemma *rank-lt-Un-rank-eq*: $\text{rank-lt } x \cup \text{rank-eq } x = \text{rank-le } x$
 ⟨proof⟩

21.4.4 Sequencing basis elements

definition

place :: 'a compact-basis \Rightarrow nat

where

place $x = \text{card } (\text{rank-lt } x) + \text{choose-pos } (\text{rank-eq } x) x$

lemma *place-bounded*: $\text{place } x < \text{card } (\text{rank-le } x)$
 ⟨proof⟩

lemma *place-ge*: $\text{card } (\text{rank-lt } x) \leq \text{place } x$
 ⟨proof⟩

lemma *place-rank-mono*:

fixes $x y$:: 'a compact-basis

shows $\text{rank } x < \text{rank } y \Longrightarrow \text{place } x < \text{place } y$

⟨proof⟩

lemma *place-eqD*: $\text{place } x = \text{place } y \Longrightarrow x = y$
 ⟨proof⟩

lemma *inj-place*: $\text{inj } \text{place}$

⟨proof⟩

21.4.5 Embedding and projection on basis elements

definition

sub :: 'a compact-basis \Rightarrow 'a compact-basis

where

$sub\ x = (case\ rank\ x\ of\ 0 \Rightarrow compact-bot \mid Suc\ k \Rightarrow cb-take\ k\ x)$

lemma *rank-sub-less*: $x \neq compact-bot \implies rank\ (sub\ x) < rank\ x$
 ⟨proof⟩

lemma *place-sub-less*: $x \neq compact-bot \implies place\ (sub\ x) < place\ x$
 ⟨proof⟩

lemma *sub-below*: $sub\ x \sqsubseteq x$
 ⟨proof⟩

lemma *rank-less-imp-below-sub*: $\llbracket x \sqsubseteq y; rank\ x < rank\ y \rrbracket \implies x \sqsubseteq sub\ y$
 ⟨proof⟩

function *basis-emb* :: 'a compact-basis \Rightarrow ubasis
 where *basis-emb* $x = (if\ x = compact-bot\ then\ 0\ else$
 node (place x) (*basis-emb* (sub x))
 (*basis-emb* ‘ { $y.$ place $y < place\ x \wedge x \sqsubseteq y$ }))
 ⟨proof⟩

termination *basis-emb*
 ⟨proof⟩

declare *basis-emb.simps* [simp del]

lemma *basis-emb-compact-bot* [simp]:
basis-emb compact-bot = 0
 ⟨proof⟩

lemma *basis-emb-rec*:
basis-emb = *node* (place x) (*basis-emb* (sub x)) (*basis-emb* ‘ { $y.$ place $y < place\ x \wedge x \sqsubseteq y$ }))
if $x \neq compact-bot$
 ⟨proof⟩

lemma *basis-emb-eq-0-iff* [simp]:
basis-emb $x = 0 \iff x = compact-bot$
 ⟨proof⟩

lemma *fin1*: finite { $y.$ place $y < place\ x \wedge x \sqsubseteq y$ }
 ⟨proof⟩

lemma *fin2*: finite (*basis-emb* ‘ { $y.$ place $y < place\ x \wedge x \sqsubseteq y$ }))
 ⟨proof⟩

lemma *rank-place-mono*:
 $\llbracket place\ x < place\ y; x \sqsubseteq y \rrbracket \implies rank\ x < rank\ y$
 ⟨proof⟩

lemma *basis-emb-mono*:

$x \sqsubseteq y \implies \text{ubasis-le } (\text{basis-emb } x) (\text{basis-emb } y)$
 ⟨proof⟩

lemma *inj-basis-emb*: *inj basis-emb*

⟨proof⟩

definition

basis-prj :: *ubasis* \Rightarrow 'a *compact-basis*

where

basis-prj *x* = *inv basis-emb*

(*ubasis-until* ($\lambda x. x \in \text{range } (\text{basis-emb} :: 'a \text{ compact-basis} \Rightarrow \text{ubasis})$) *x*)

lemma *basis-prj-basis-emb*: $\bigwedge x. \text{basis-prj } (\text{basis-emb } x) = x$

⟨proof⟩

lemma *basis-prj-node*:

$\llbracket \text{finite } S; \text{node } i \text{ a } S \notin \text{range } (\text{basis-emb} :: 'a \text{ compact-basis} \Rightarrow \text{nat}) \rrbracket$

$\implies \text{basis-prj } (\text{node } i \text{ a } S) = (\text{basis-prj } a :: 'a \text{ compact-basis})$

⟨proof⟩

lemma *basis-prj-0*: *basis-prj* 0 = *compact-bot*

⟨proof⟩

lemma *node-eq-basis-emb-iff*:

finite *S* $\implies \text{node } i \text{ a } S = \text{basis-emb } x \iff$

$x \neq \text{compact-bot} \wedge i = \text{place } x \wedge a = \text{basis-emb } (\text{sub } x) \wedge$

$S = \text{basis-emb } \{ y. \text{place } y < \text{place } x \wedge x \sqsubseteq y \}$

⟨proof⟩

lemma *basis-prj-mono*: *ubasis-le* *a* *b* $\implies \text{basis-prj } a \sqsubseteq \text{basis-prj } b$

⟨proof⟩

lemma *basis-emb-prj-less*: *ubasis-le* (*basis-emb* (*basis-prj* *x*)) *x*

⟨proof⟩

lemma *ideal-completion*:

ideal-completion *below* *Rep-compact-basis* (*approximants* :: 'a \Rightarrow -)

⟨proof⟩

end

interpretation *compact-basis*:

ideal-completion *below* *Rep-compact-basis*

approximants :: 'a::bifinite \Rightarrow 'a *compact-basis* *set*

⟨proof⟩

21.4.6 EP-pair from any bifinite domain into *u*dom**context** *bifinite-approx-chain* **begin****definition***u*dom-emb :: 'a → *u*dom**where***u*dom-emb = compact-basis.extension (λx. *u*dom-principal (basis-emb x))**definition***u*dom-prj :: *u*dom → 'a**where***u*dom-prj = *u*dom.extension (λx. Rep-compact-basis (basis-prj x))**lemma** *u*dom-emb-principal:*u*dom-emb · (Rep-compact-basis x) = *u*dom-principal (basis-emb x)

⟨proof⟩

lemma *u*dom-prj-principal:*u*dom-prj · (*u*dom-principal x) = Rep-compact-basis (basis-prj x)

⟨proof⟩

lemma *ep-pair-u*dom: *ep-pair u*dom-emb *u*dom-prj

⟨proof⟩

end**abbreviation** *u*dom-emb ≡ *bifinite-approx-chain.u*dom-emb**abbreviation** *u*dom-prj ≡ *bifinite-approx-chain.u*dom-prj**lemmas** *ep-pair-u*dom =*bifinite-approx-chain.ep-pair-u*dom [unfolded *bifinite-approx-chain-def*]**21.5 Chain of approx functions for type *u*dom****definition***u*dom-approx :: nat ⇒ *u*dom → *u*dom**where***u*dom-approx i =*u*dom.extension (λx. *u*dom-principal (ubasis-until (λy. y ≤ i) x))**lemma** *u*dom-approx-mono:

ubasis-le a b ⇒

*u*dom-principal (ubasis-until (λy. y ≤ i) a) ⊆*u*dom-principal (ubasis-until (λy. y ≤ i) b)

⟨proof⟩

lemma *adm-mem-finite*: [cont f; finite S] ⇒ adm (λx. f x ∈ S)

⟨proof⟩

lemma *udom-approx-principal*:
 $udom-approx\ i \cdot (udom-principal\ x) =$
 $udom-principal\ (ubasis-until\ (\lambda y. y \leq i)\ x)$
 $\langle proof \rangle$

lemma *finite-deflation-udom-approx*: $finite-deflation\ (udom-approx\ i)$
 $\langle proof \rangle$

interpretation *udom-approx*: $finite-deflation\ udom-approx\ i$
 $\langle proof \rangle$

lemma *chain-udom-approx* [*simp*]: $chain\ (\lambda i. udom-approx\ i)$
 $\langle proof \rangle$

lemma *lub-udom-approx* [*simp*]: $(\bigsqcup i. udom-approx\ i) = ID$
 $\langle proof \rangle$

lemma *udom-approx* [*simp*]: $approx-chain\ udom-approx$
 $\langle proof \rangle$

instance *udom* :: *bifinite*
 $\langle proof \rangle$

hide-const (**open**) *node*

notation *binomial* (**infixl** *choose* 65)

end

22 Algebraic deflations

theory *Algebraic*
imports *Universal Map-Functions*
begin

default-sort *bifinite*

22.1 Type constructor for finite deflations

typedef $'a\ fin-defl = \{d :: 'a \rightarrow 'a. finite-deflation\ d\}$
 $\langle proof \rangle$

instantiation *fin-defl* :: (*bifinite*) *below*
begin

definition *below-fin-defl-def*:
 $below \equiv \lambda x\ y. Rep-fin-defl\ x \sqsubseteq Rep-fin-defl\ y$

instance $\langle proof \rangle$

end

instance *fin-defl* :: (*bifinite*) *po*
 ⟨*proof*⟩

lemma *finite-deflation-Rep-fin-defl*: *finite-deflation* (*Rep-fin-defl d*)
 ⟨*proof*⟩

lemma *deflation-Rep-fin-defl*: *deflation* (*Rep-fin-defl d*)
 ⟨*proof*⟩

interpretation *Rep-fin-defl*: *finite-deflation* *Rep-fin-defl d*
 ⟨*proof*⟩

lemma *fin-defl-belowI*:
 $(\bigwedge x. \text{Rep-fin-defl } a \cdot x = x \implies \text{Rep-fin-defl } b \cdot x = x) \implies a \sqsubseteq b$
 ⟨*proof*⟩

lemma *fin-defl-belowD*:
 $\llbracket a \sqsubseteq b; \text{Rep-fin-defl } a \cdot x = x \rrbracket \implies \text{Rep-fin-defl } b \cdot x = x$
 ⟨*proof*⟩

lemma *fin-defl-eqI*:
 $(\bigwedge x. \text{Rep-fin-defl } a \cdot x = x \iff \text{Rep-fin-defl } b \cdot x = x) \implies a = b$
 ⟨*proof*⟩

lemma *Rep-fin-defl-mono*: $a \sqsubseteq b \implies \text{Rep-fin-defl } a \sqsubseteq \text{Rep-fin-defl } b$
 ⟨*proof*⟩

lemma *Abs-fin-defl-mono*:
 $\llbracket \text{finite-deflation } a; \text{finite-deflation } b; a \sqsubseteq b \rrbracket$
 $\implies \text{Abs-fin-defl } a \sqsubseteq \text{Abs-fin-defl } b$
 ⟨*proof*⟩

lemma (**in** *finite-deflation*) *compact-belowI*:
assumes $\bigwedge x. \text{compact } x \implies d \cdot x = x \implies f \cdot x = x$ **shows** $d \sqsubseteq f$
 ⟨*proof*⟩

lemma *compact-Rep-fin-defl* [*simp*]: *compact* (*Rep-fin-defl a*)
 ⟨*proof*⟩

22.2 Defining algebraic deflations by ideal completion

typedef *'a defl* = $\{S :: 'a \text{ fin-defl set. below.ideal } S\}$
 ⟨*proof*⟩

instantiation *defl* :: (*bifinite*) *below*
begin

definition

$$x \sqsubseteq y \longleftrightarrow \text{Rep-defl } x \subseteq \text{Rep-defl } y$$
instance $\langle \text{proof} \rangle$ **end****instance** $\text{defl} :: (\text{bifinite}) \text{ po}$ $\langle \text{proof} \rangle$ **instance** $\text{defl} :: (\text{bifinite}) \text{ cpo}$ $\langle \text{proof} \rangle$ **definition**

$$\begin{aligned} \text{defl-principal} &:: 'a \text{ fin-defl} \Rightarrow 'a \text{ defl} \textbf{ where} \\ \text{defl-principal } t &= \text{Abs-defl } \{u. u \sqsubseteq t\} \end{aligned}$$
lemma $\text{fin-defl-countable}: \exists f :: 'a \text{ fin-defl} \Rightarrow \text{nat. inj } f$ $\langle \text{proof} \rangle$ **interpretation** $\text{defl}: \text{ideal-completion below defl-principal Rep-defl}$ $\langle \text{proof} \rangle$

Algebraic deflations are pointed

lemma $\text{defl-minimal}: \text{defl-principal } (\text{Abs-fin-defl } \perp) \sqsubseteq x$ $\langle \text{proof} \rangle$ **instance** $\text{defl} :: (\text{bifinite}) \text{ pcpo}$ $\langle \text{proof} \rangle$ **lemma** $\text{inst-defl-pcpo}: \perp = \text{defl-principal } (\text{Abs-fin-defl } \perp)$ $\langle \text{proof} \rangle$

22.3 Applying algebraic deflations

definition

$$\text{cast} :: 'a \text{ defl} \rightarrow 'a \rightarrow 'a$$
where

$$\text{cast} = \text{defl.extension Rep-fin-defl}$$
lemma $\text{cast-defl-principal}:$

$$\text{cast} \cdot (\text{defl-principal } a) = \text{Rep-fin-defl } a$$
 $\langle \text{proof} \rangle$ **lemma** $\text{deflation-cast}: \text{deflation } (\text{cast} \cdot d)$ $\langle \text{proof} \rangle$ **lemma** $\text{finite-deflation-cast}:$

$$\text{compact } d \implies \text{finite-deflation } (\text{cast} \cdot d)$$
 $\langle \text{proof} \rangle$

interpretation *cast*: deflation *cast*·*d*

<proof>

declare *cast.idem* [*simp*]

lemma *compact-cast* [*simp*]: *compact d* \implies *compact (cast*·*d)*

<proof>

lemma *cast-below-cast*: *cast*·*A* \sqsubseteq *cast*·*B* \longleftrightarrow *A* \sqsubseteq *B*

<proof>

lemma *compact-cast-iff*: *compact (cast*·*d)* \longleftrightarrow *compact d*

<proof>

lemma *cast-below-imp-below*: *cast*·*A* \sqsubseteq *cast*·*B* \implies *A* \sqsubseteq *B*

<proof>

lemma *cast-eq-imp-eq*: *cast*·*A* = *cast*·*B* \implies *A* = *B*

<proof>

lemma *cast-strict1* [*simp*]: *cast*· \perp = \perp

<proof>

lemma *cast-strict2* [*simp*]: *cast*·*A*· \perp = \perp

<proof>

22.4 Deflation combinators

definition

$$\begin{aligned} \text{defl-fun1 } e \text{ } p \text{ } f = & \\ & \text{defl.extension } (\lambda a. \\ & \text{defl-principal } (\text{Abs-fin-defl} \\ & (e \text{ oo } f \cdot (\text{Rep-fin-defl } a) \text{ oo } p))) \end{aligned}$$

definition

$$\begin{aligned} \text{defl-fun2 } e \text{ } p \text{ } f = & \\ & \text{defl.extension } (\lambda a. \\ & \text{defl.extension } (\lambda b. \\ & \text{defl-principal } (\text{Abs-fin-defl} \\ & (e \text{ oo } f \cdot (\text{Rep-fin-defl } a) \cdot (\text{Rep-fin-defl } b) \text{ oo } p)))) \end{aligned}$$

lemma *cast-defl-fun1*:

assumes *ep*: *ep-pair e p*

assumes *f*: $\bigwedge a. \text{finite-deflation } a \implies \text{finite-deflation } (f \cdot a)$

shows *cast*·(*defl-fun1 e p f*·*A*) = *e* oo *f*·(*cast*·*A*) oo *p*

<proof>

lemma *cast-defl-fun2*:

```

assumes ep: ep-pair e p
assumes f:  $\bigwedge a b.$  finite-deflation a  $\implies$  finite-deflation b  $\implies$ 
           finite-deflation (f·a·b)
shows cast·(defl-fun2 e p f·A·B) = e oo f·(cast·A)·(cast·B) oo p
⟨proof⟩

end

```

23 Representable domains

```

theory Representable
imports Algebraic Map-Functions HOL-Library.Countable
begin

```

```

default-sort cpo

```

23.1 Class of representable domains

We define a “domain” as a pcpo that is isomorphic to some algebraic deflation over the universal domain; this is equivalent to being omega-bifinite.

A predomain is a cpo that, when lifted, becomes a domain. Predomains are represented by deflations over a lifted universal domain type.

```

class predomain-syn = cpo +
  fixes liftemb :: 'a⊥ → udom⊥
  fixes liftprj :: udom⊥ → 'a⊥
  fixes liftdefl :: 'a itself ⇒ udom u defl

class predomain = predomain-syn +
  assumes predomain-ep: ep-pair liftemb liftprj
  assumes cast-liftdefl: cast·(liftdefl TYPE('a)) = liftemb oo liftprj

syntax -LIFTDEFL :: type ⇒ logic ((1LIFTDEFL/(1'(-))))
translations LIFTDEFL('t) ⇔ CONST liftdefl TYPE('t)

```

```

definition liftdefl-of :: udom defl → udom u defl
  where liftdefl-of = defl-fun1 ID ID u-map

```

```

lemma cast-liftdefl-of: cast·(liftdefl-of·t) = u-map·(cast·t)
⟨proof⟩

```

```

class domain = predomain-syn + pcpo +
  fixes emb :: 'a → udom
  fixes prj :: udom → 'a
  fixes defl :: 'a itself ⇒ udom defl
  assumes ep-pair-emb-prj: ep-pair emb prj
  assumes cast-DEFL: cast·(defl TYPE('a)) = emb oo prj
  assumes liftemb-eq: liftemb = u-map·emb
  assumes liftprj-eq: liftprj = u-map·prj

```

assumes *liftdefl-eq*: $\text{liftdefl } \text{TYPE}('a) = \text{liftdefl-of} \cdot (\text{defl } \text{TYPE}('a))$

syntax *-DEFL* :: $\text{type} \Rightarrow \text{logic } ((1\text{DEFL}/(1'(-'))))$
translations $\text{DEFL}(t) \Leftrightarrow \text{CONST defl } \text{TYPE}(t)$

instance $\text{domain} \subseteq \text{predomain}$
<proof>

Constants *liftemb* and *liftprj* imply class *predomain*.

<ML>

interpretation *predomain*: *pcpo-ep-pair liftemb liftprj*
<proof>

interpretation *domain*: *pcpo-ep-pair emb prj*
<proof>

lemmas *emb-inverse* = *domain.e-inverse*

lemmas *emb-prj-below* = *domain.e-p-below*

lemmas *emb-eq-iff* = *domain.e-eq-iff*

lemmas *emb-strict* = *domain.e-strict*

lemmas *prj-strict* = *domain.p-strict*

23.2 Domains are bifinite

lemma *approx-chain-ep-cast*:

assumes *ep*: *ep-pair* ($e::'a::\text{pcpo} \rightarrow 'b::\text{bifinite}$) ($p::'b \rightarrow 'a$)

assumes *cast-t*: $\text{cast} \cdot t = e \text{ oo } p$

shows $\exists (a::\text{nat} \Rightarrow 'a::\text{pcpo} \rightarrow 'a)$. *approx-chain a*

<proof>

instance $\text{domain} \subseteq \text{bifinite}$
<proof>

instance $\text{predomain} \subseteq \text{profinite}$
<proof>

23.3 Universal domain ep-pairs

definition *u-emb* = *udom-emb* ($\lambda i. \text{u-map} \cdot (\text{udom-approx } i)$)

definition *u-prj* = *udom-prj* ($\lambda i. \text{u-map} \cdot (\text{udom-approx } i)$)

definition *prod-emb* = *udom-emb* ($\lambda i. \text{prod-map} \cdot (\text{udom-approx } i) \cdot (\text{udom-approx } i)$)

definition *prod-prj* = *udom-prj* ($\lambda i. \text{prod-map} \cdot (\text{udom-approx } i) \cdot (\text{udom-approx } i)$)

definition *sprod-emb* = *udom-emb* ($\lambda i. \text{sprod-map} \cdot (\text{udom-approx } i) \cdot (\text{udom-approx } i)$)

definition *sprod-prj* = *udom-prj* ($\lambda i. \text{sprod-map} \cdot (\text{udom-approx } i) \cdot (\text{udom-approx } i)$)

definition $ssum-emb = udom-emb (\lambda i. ssum-map \cdot (udom-approx\ i) \cdot (udom-approx\ i))$

definition $ssum-prj = udom-prj (\lambda i. ssum-map \cdot (udom-approx\ i) \cdot (udom-approx\ i))$

definition $sfun-emb = udom-emb (\lambda i. sfun-map \cdot (udom-approx\ i) \cdot (udom-approx\ i))$

definition $sfun-prj = udom-prj (\lambda i. sfun-map \cdot (udom-approx\ i) \cdot (udom-approx\ i))$

lemma $ep-pair-u: ep-pair\ u-emb\ u-prj$

$\langle proof \rangle$

lemma $ep-pair-prod: ep-pair\ prod-emb\ prod-prj$

$\langle proof \rangle$

lemma $ep-pair-sprod: ep-pair\ sprod-emb\ sprod-prj$

$\langle proof \rangle$

lemma $ep-pair-ssum: ep-pair\ ssum-emb\ ssum-prj$

$\langle proof \rangle$

lemma $ep-pair-sfun: ep-pair\ sfun-emb\ sfun-prj$

$\langle proof \rangle$

23.4 Type combinators

definition $u-defl :: udom\ defl \rightarrow udom\ defl$

where $u-defl = defl-fun1\ u-emb\ u-prj\ u-map$

definition $prod-defl :: udom\ defl \rightarrow udom\ defl \rightarrow udom\ defl$

where $prod-defl = defl-fun2\ prod-emb\ prod-prj\ prod-map$

definition $sprod-defl :: udom\ defl \rightarrow udom\ defl \rightarrow udom\ defl$

where $sprod-defl = defl-fun2\ sprod-emb\ sprod-prj\ sprod-map$

definition $ssum-defl :: udom\ defl \rightarrow udom\ defl \rightarrow udom\ defl$

where $ssum-defl = defl-fun2\ ssum-emb\ ssum-prj\ ssum-map$

definition $sfun-defl :: udom\ defl \rightarrow udom\ defl \rightarrow udom\ defl$

where $sfun-defl = defl-fun2\ sfun-emb\ sfun-prj\ sfun-map$

lemma $cast-u-defl:$

$cast \cdot (u-defl \cdot A) = u-emb\ oo\ u-map \cdot (cast \cdot A)\ oo\ u-prj$

$\langle proof \rangle$

lemma $cast-prod-defl:$

$cast \cdot (prod-defl \cdot A \cdot B) =$

$prod-emb\ oo\ prod-map \cdot (cast \cdot A) \cdot (cast \cdot B)\ oo\ prod-prj$

$\langle proof \rangle$

lemma *cast-sprod-defl*:

$$\begin{aligned} \text{cast} \cdot (\text{sprod-defl} \cdot A \cdot B) &= \\ \text{sprod-emb} \text{ oo } \text{sprod-map} \cdot (\text{cast} \cdot A) \cdot (\text{cast} \cdot B) \text{ oo } \text{sprod-prj} \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *cast-ssum-defl*:

$$\begin{aligned} \text{cast} \cdot (\text{ssum-defl} \cdot A \cdot B) &= \\ \text{ssum-emb} \text{ oo } \text{ssum-map} \cdot (\text{cast} \cdot A) \cdot (\text{cast} \cdot B) \text{ oo } \text{ssum-prj} \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *cast-sfun-defl*:

$$\begin{aligned} \text{cast} \cdot (\text{sfun-defl} \cdot A \cdot B) &= \\ \text{sfun-emb} \text{ oo } \text{sfun-map} \cdot (\text{cast} \cdot A) \cdot (\text{cast} \cdot B) \text{ oo } \text{sfun-prj} \\ \langle \text{proof} \rangle \end{aligned}$$

Special deflation combinator for unpointed types.

definition *u-liftdefl* :: *udom u defl* → *udom defl*
where *u-liftdefl* = *defl-fun1 u-emb u-prj ID*

lemma *cast-u-liftdefl*:

$$\begin{aligned} \text{cast} \cdot (\text{u-liftdefl} \cdot A) &= \text{u-emb} \text{ oo } \text{cast} \cdot A \text{ oo } \text{u-prj} \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *u-liftdefl-liftdefl-of*:

$$\begin{aligned} \text{u-liftdefl} \cdot (\text{liftdefl-of} \cdot A) &= \text{u-defl} \cdot A \\ \langle \text{proof} \rangle \end{aligned}$$

23.5 Class instance proofs

23.5.1 Universal domain

instantiation *udom* :: *domain*
begin

definition [*simp*]:

$$\text{emb} = (\text{ID} :: \text{udom} \rightarrow \text{udom})$$

definition [*simp*]:

$$\text{prj} = (\text{ID} :: \text{udom} \rightarrow \text{udom})$$

definition

$$\text{defl} (t :: \text{udom itself}) = (\bigsqcup i. \text{defl-principal} (\text{Abs-fin-defl} (\text{udom-approx } i)))$$

definition

$$(\text{liftemb} :: \text{udom } u \rightarrow \text{udom } u) = \text{u-map} \cdot \text{emb}$$

definition

$$(\text{liftprj} :: \text{udom } u \rightarrow \text{udom } u) = \text{u-map} \cdot \text{prj}$$

definition

$liftdefl (t::udom \textit{ itself}) = liftdefl\text{-of}\cdot DEFL(udom)$

instance $\langle proof \rangle$

end

23.5.2 Lifted cpo

instantiation $u :: (predomain) \textit{ domain}$
begin

definition

$emb = u\text{-emb} \textit{ oo} liftemb$

definition

$prj = liftprj \textit{ oo} u\text{-prj}$

definition

$defl (t::'a \textit{ u} \textit{ itself}) = u\text{-liftdefl}\cdot LIFTDEFL('a)$

definition

$(liftemb :: 'a \textit{ u} \textit{ u} \rightarrow udom \textit{ u}) = u\text{-map}\cdot emb$

definition

$(liftprj :: udom \textit{ u} \rightarrow 'a \textit{ u} \textit{ u}) = u\text{-map}\cdot prj$

definition

$liftdefl (t::'a \textit{ u} \textit{ itself}) = liftdefl\text{-of}\cdot DEFL('a \textit{ u})$

instance $\langle proof \rangle$

end

lemma $DEFL\text{-}u$: $DEFL('a::predomain \textit{ u}) = u\text{-liftdefl}\cdot LIFTDEFL('a)$
 $\langle proof \rangle$

23.5.3 Strict function space

instantiation $sfun :: (domain, domain) \textit{ domain}$
begin

definition

$emb = sfun\text{-emb} \textit{ oo} sfun\text{-map}\cdot prj\cdot emb$

definition

$prj = sfun\text{-map}\cdot emb\cdot prj \textit{ oo} sfun\text{-prj}$

definition

$defl (t::('a \rightarrow! 'b) \textit{ itself}) = sfun\text{-defl}\cdot DEFL('a)\cdot DEFL('b)$

definition

$$(liftemb :: ('a \to! 'b) u \to udom\ u) = u\text{-map}\cdot emb$$
definition

$$(liftprj :: udom\ u \to ('a \to! 'b) u) = u\text{-map}\cdot prj$$
definition

$$liftdefl\ (t :: ('a \to! 'b) itself) = liftdefl\text{-of}\cdot DEFL('a \to! 'b)$$
instance $\langle proof \rangle$ **end****lemma** *DEFL-sfun*:
$$DEFL('a :: domain \to! 'b :: domain) = sfun\text{-defl}\cdot DEFL('a)\cdot DEFL('b)$$

$$\langle proof \rangle$$
23.5.4 Continuous function space**instantiation** *cfun* :: $(pre\ domain, domain) domain$ **begin****definition**

$$emb = emb\ oo\ encode\text{-}cfun$$
definition

$$prj = decode\text{-}cfun\ oo\ prj$$
definition

$$defl\ (t :: ('a \to 'b) itself) = DEFL('a\ u \to! 'b)$$
definition

$$(liftemb :: ('a \to 'b) u \to udom\ u) = u\text{-map}\cdot emb$$
definition

$$(liftprj :: udom\ u \to ('a \to 'b) u) = u\text{-map}\cdot prj$$
definition

$$liftdefl\ (t :: ('a \to 'b) itself) = liftdefl\text{-of}\cdot DEFL('a \to 'b)$$
instance $\langle proof \rangle$ **end****lemma** *DEFL-cfun*:
$$DEFL('a :: pre\ domain \to 'b :: domain) = DEFL('a\ u \to! 'b)$$

$$\langle proof \rangle$$

23.5.5 Strict product

instantiation *sprod* :: (*domain*, *domain*) *domain*
begin

definition

$$emb = sprod-emb \text{ oo } sprod-map \cdot emb \cdot emb$$

definition

$$prj = sprod-map \cdot prj \cdot prj \text{ oo } sprod-prj$$

definition

$$defl (t :: ('a \otimes 'b) \text{ itself}) = sprod-defl \cdot DEFL('a) \cdot DEFL('b)$$

definition

$$(liftemb :: ('a \otimes 'b) \text{ } u \rightarrow \text{ } u \text{ dom } u) = u-map \cdot emb$$

definition

$$(liftprj :: \text{ } u \text{ dom } u \rightarrow ('a \otimes 'b) \text{ } u) = u-map \cdot prj$$

definition

$$liftdefl (t :: ('a \otimes 'b) \text{ itself}) = liftdefl-of \cdot DEFL('a \otimes 'b)$$

instance $\langle proof \rangle$

end

lemma *DEFL-sprod*:

$$DEFL('a :: \text{domain} \otimes 'b :: \text{domain}) = sprod-defl \cdot DEFL('a) \cdot DEFL('b)$$

$\langle proof \rangle$

23.5.6 Cartesian product

definition *prod-liftdefl* :: *u dom u defl* \rightarrow *u dom u defl* \rightarrow *u dom u defl*
where *prod-liftdefl* = *defl-fun2* (*u-map* \cdot *prod-emb* *oo* *decode-prod-u*)
(*encode-prod-u* *oo* *u-map* \cdot *prod-prj*) *sprod-map*

lemma *cast-prod-liftdefl*:

$$cast \cdot (prod-liftdefl \cdot a \cdot b) =$$

$$(u-map \cdot prod-emb \text{ oo } decode-prod-u) \text{ oo } sprod-map \cdot (cast \cdot a) \cdot (cast \cdot b) \text{ oo}$$

$$(encode-prod-u \text{ oo } u-map \cdot prod-prj)$$

$\langle proof \rangle$

instantiation *prod* :: (*predomain*, *predomain*) *predomain*
begin

definition

$$liftemb = (u-map \cdot prod-emb \text{ oo } decode-prod-u) \text{ oo}$$

$$(sprod-map \cdot liftemb \cdot liftemb \text{ oo } encode-prod-u)$$

definition

$$\text{liftprj} = (\text{decode-prod-u} \text{ oo } \text{sprod-map} \cdot \text{liftprj} \cdot \text{liftprj}) \text{ oo}$$

$$(\text{encode-prod-u} \text{ oo } \text{u-map} \cdot \text{prod-prj})$$
definition

$$\text{liftdefl} (t :: ('a \times 'b) \text{ itself}) = \text{prod-liftdefl} \cdot \text{LIFTDEFL}('a) \cdot \text{LIFTDEFL}('b)$$
instance $\langle \text{proof} \rangle$ **end****instantiation** $\text{prod} :: (\text{domain}, \text{domain}) \text{ domain}$ **begin****definition**

$$\text{emb} = \text{prod-emb} \text{ oo } \text{prod-map} \cdot \text{emb} \cdot \text{emb}$$
definition

$$\text{prj} = \text{prod-map} \cdot \text{prj} \cdot \text{prj} \text{ oo } \text{prod-prj}$$
definition

$$\text{defl} (t :: ('a \times 'b) \text{ itself}) = \text{prod-defl} \cdot \text{DEFL}('a) \cdot \text{DEFL}('b)$$
instance $\langle \text{proof} \rangle$ **end****lemma** *DEFL-prod*:
$$\text{DEFL}('a :: \text{domain} \times 'b :: \text{domain}) = \text{prod-defl} \cdot \text{DEFL}('a) \cdot \text{DEFL}('b)$$

$$\langle \text{proof} \rangle$$
lemma *LIFTDEFL-prod*:
$$\text{LIFTDEFL}('a :: \text{predomain} \times 'b :: \text{predomain}) =$$

$$\text{prod-liftdefl} \cdot \text{LIFTDEFL}('a) \cdot \text{LIFTDEFL}('b)$$

$$\langle \text{proof} \rangle$$
23.5.7 Unit type**instantiation** $\text{unit} :: \text{domain}$ **begin****definition**

$$\text{emb} = (\perp :: \text{unit} \rightarrow \text{udom})$$
definition

$$\text{prj} = (\perp :: \text{udom} \rightarrow \text{unit})$$
definition

$$\text{defl} (t :: \text{unit} \text{ itself}) = \perp$$

definition

$$(liftemb :: unit\ u \rightarrow udom\ u) = u\text{-map}\cdot emb$$
definition

$$(liftprj :: udom\ u \rightarrow unit\ u) = u\text{-map}\cdot prj$$
definition

$$liftdefl\ (t :: unit\ itself) = liftdefl\text{-of}\cdot DEFL\ (unit)$$

instance $\langle proof \rangle$

end

23.5.8 Discrete cpo

instantiation $discr :: (countable)\ predomain$

begin

definition

$$(liftemb :: 'a\ discr\ u \rightarrow udom\ u) = strictify\text{-up}\ oo\ udom\text{-emb}\ discr\text{-approx}$$
definition

$$(liftprj :: udom\ u \rightarrow 'a\ discr\ u) = udom\text{-prj}\ discr\text{-approx}\ oo\ fup\cdot ID$$
definition

$$liftdefl\ (t :: 'a\ discr\ itself) =$$

$$(\bigsqcup\ i.\ defl\text{-principal}\ (Abs\text{-fin}\text{-defl}\ (liftemb\ oo\ discr\text{-approx}\ i\ oo\ (liftprj :: udom\ u \rightarrow 'a\ discr\ u))))$$

instance $\langle proof \rangle$

end

23.5.9 Strict sum

instantiation $ssum :: (domain,\ domain)\ domain$

begin

definition

$$emb = ssum\text{-emb}\ oo\ ssum\text{-map}\cdot emb\cdot emb$$
definition

$$prj = ssum\text{-map}\cdot prj\cdot prj\ oo\ ssum\text{-prj}$$
definition

$$defl\ (t :: ('a \oplus 'b)\ itself) = ssum\text{-defl}\cdot DEFL\ ('a)\cdot DEFL\ ('b)$$
definition

$$(liftemb :: ('a \oplus 'b)\ u \rightarrow udom\ u) = u\text{-map}\cdot emb$$

definition

$$(liftprj :: udom\ u \rightarrow ('a \oplus 'b)\ u) = u\text{-map}\cdot prj$$
definition

$$liftdefl\ (t :: ('a \oplus 'b)\ itself) = liftdefl\text{-of}\cdot DEFL('a \oplus 'b)$$
instance $\langle proof \rangle$ **end****lemma** *DEFL-ssum*:
$$DEFL('a :: domain \oplus 'b :: domain) = ssum\text{-defl}\cdot DEFL('a)\cdot DEFL('b)$$

$$\langle proof \rangle$$
23.5.10 Lifted HOL type**instantiation** *lift* :: (countable) domain**begin****definition**

$$emb = emb\ oo\ (\Lambda\ x.\ Rep\text{-lift}\ x)$$
definition

$$prj = (\Lambda\ y.\ Abs\text{-lift}\ y)\ oo\ prj$$
definition

$$defl\ (t :: 'a\ lift\ itself) = DEFL('a\ discr\ u)$$
definition

$$(liftemb :: 'a\ lift\ u \rightarrow udom\ u) = u\text{-map}\cdot emb$$
definition

$$(liftprj :: udom\ u \rightarrow 'a\ lift\ u) = u\text{-map}\cdot prj$$
definition

$$liftdefl\ (t :: 'a\ lift\ itself) = liftdefl\text{-of}\cdot DEFL('a\ lift)$$
instance $\langle proof \rangle$ **end****end****24 The unit domain****theory** *One* **imports** *Lift***begin**

type-synonym $one = unit\ lift$

translations

$(type)\ one \leftarrow (type)\ unit\ lift$

definition $ONE :: one$

where $ONE \equiv Def\ ()$

Exhaustion and Elimination for type one

lemma $Exh-one: t = \perp \vee t = ONE$

$\langle proof \rangle$

lemma $oneE [case-names\ bottom\ ONE]: \llbracket p = \perp \implies Q; p = ONE \implies Q \rrbracket \implies Q$

$\langle proof \rangle$

lemma $one-induct [case-names\ bottom\ ONE]: P\ \perp \implies P\ ONE \implies P\ x$

$\langle proof \rangle$

lemma $dist-below-one [simp]: ONE \not\sqsubseteq \perp$

$\langle proof \rangle$

lemma $below-ONE [simp]: x \sqsubseteq ONE$

$\langle proof \rangle$

lemma $ONE-below-iff [simp]: ONE \sqsubseteq x \longleftrightarrow x = ONE$

$\langle proof \rangle$

lemma $ONE-defined [simp]: ONE \neq \perp$

$\langle proof \rangle$

lemma $one-neq-iffs [simp]:$

$x \neq ONE \longleftrightarrow x = \perp$

$ONE \neq x \longleftrightarrow x = \perp$

$x \neq \perp \longleftrightarrow x = ONE$

$\perp \neq x \longleftrightarrow x = ONE$

$\langle proof \rangle$

lemma $compact-ONE: compact\ ONE$

$\langle proof \rangle$

Case analysis function for type one

definition $one-case :: 'a::pcpo \rightarrow one \rightarrow 'a$

where $one-case = (\Lambda a\ x. seq\cdot x\cdot a)$

translations

$case\ x\ of\ XCONST\ ONE \Rightarrow t \Leftrightarrow CONST\ one-case\cdot t\cdot x$

$case\ x\ of\ XCONST\ ONE :: 'a \Rightarrow t \rightarrow CONST\ one-case\cdot t\cdot x$

$\Lambda (XCONST\ ONE). t \Leftrightarrow CONST\ one-case\cdot t$

lemma *one-case1* [simp]: (case \perp of ONE \Rightarrow t) = \perp
 ⟨proof⟩

lemma *one-case2* [simp]: (case ONE of ONE \Rightarrow t) = t
 ⟨proof⟩

lemma *one-case3* [simp]: (case x of ONE \Rightarrow ONE) = x
 ⟨proof⟩

end

25 Fixed point operator and admissibility

theory *Fix*
 imports *Cfun*
 begin

default-sort *pcpo*

25.1 Iteration

primrec *iterate* :: nat \Rightarrow ('a::cpo \rightarrow 'a) \rightarrow ('a \rightarrow 'a)
 where
 iterate 0 = (Λ F x. x)
 | *iterate* (Suc n) = (Λ F x. F·(*iterate* n·F·x))

Derive inductive properties of *iterate* from primitive recursion

lemma *iterate-0* [simp]: *iterate* 0·F·x = x
 ⟨proof⟩

lemma *iterate-Suc* [simp]: *iterate* (Suc n)·F·x = F·(*iterate* n·F·x)
 ⟨proof⟩

declare *iterate.simps* [simp del]

lemma *iterate-Suc2*: *iterate* (Suc n)·F·x = *iterate* n·F·(F·x)
 ⟨proof⟩

lemma *iterate-iterate*: *iterate* m·F·(*iterate* n·F·x) = *iterate* (m + n)·F·x
 ⟨proof⟩

The sequence of function iterations is a chain.

lemma *chain-iterate* [simp]: *chain* ($\lambda i.$ *iterate* i·F· \perp)
 ⟨proof⟩

25.2 Least fixed point operator

definition *fix* :: ('a \rightarrow 'a) \rightarrow 'a

where $fix = (\Lambda F. \lfloor i. \text{iterate } i \cdot F \cdot \perp)$

Binder syntax for fix

abbreviation $fix\text{-syn} :: ('a \Rightarrow 'a) \Rightarrow 'a$ (**binder** μ 10)

where $fix\text{-syn} (\lambda x. f x) \equiv fix \cdot (\Lambda x. f x)$

notation (*ASCII*)

$fix\text{-syn}$ (**binder** *FIX* 10)

Properties of fix

direct connection between fix and iteration

lemma $fix\text{-def2}$: $fix \cdot F = (\lfloor i. \text{iterate } i \cdot F \cdot \perp)$

<proof>

lemma $iterate\text{-below}\text{-fix}$: $\text{iterate } n \cdot f \cdot \perp \sqsubseteq fix \cdot f$

<proof>

Kleene’s fixed point theorems for continuous functions in pointed omega cpo’s

lemma $fix\text{-eq}$: $fix \cdot F = F \cdot (fix \cdot F)$

<proof>

lemma $fix\text{-least}\text{-below}$: $F \cdot x \sqsubseteq x \implies fix \cdot F \sqsubseteq x$

<proof>

lemma $fix\text{-least}$: $F \cdot x = x \implies fix \cdot F \sqsubseteq x$

<proof>

lemma $fix\text{-eqI}$:

assumes $fixed$: $F \cdot x = x$

and $least$: $\bigwedge z. F \cdot z = z \implies x \sqsubseteq z$

shows $fix \cdot F = x$

<proof>

lemma $fix\text{-eq2}$: $f \equiv fix \cdot F \implies f = F \cdot f$

<proof>

lemma $fix\text{-eq3}$: $f \equiv fix \cdot F \implies f \cdot x = F \cdot f \cdot x$

<proof>

lemma $fix\text{-eq4}$: $f = fix \cdot F \implies f = F \cdot f$

<proof>

lemma $fix\text{-eq5}$: $f = fix \cdot F \implies f \cdot x = F \cdot f \cdot x$

<proof>

strictness of fix

lemma $fix\text{-bottom}\text{-iff}$: $fix \cdot F = \perp \iff F \cdot \perp = \perp$

<proof>

lemma *fix-strict*: $F \cdot \perp = \perp \implies \text{fix} \cdot F = \perp$
<proof>

lemma *fix-defined*: $F \cdot \perp \neq \perp \implies \text{fix} \cdot F \neq \perp$
<proof>

fix applied to identity and constant functions

lemma *fix-id*: $(\mu x. x) = \perp$
<proof>

lemma *fix-const*: $(\mu x. c) = c$
<proof>

25.3 Fixed point induction

lemma *fix-ind*: $\text{adm } P \implies P \perp \implies (\bigwedge x. P x \implies P (F \cdot x)) \implies P (\text{fix} \cdot F)$
<proof>

lemma *cont-fix-ind*: $\text{cont } F \implies \text{adm } P \implies P \perp \implies (\bigwedge x. P x \implies P (F x)) \implies P (\text{fix} \cdot (\text{Abs-cfun } F))$
<proof>

lemma *def-fix-ind*: $\llbracket f \equiv \text{fix} \cdot F; \text{adm } P; P \perp; \bigwedge x. P x \implies P (F \cdot x) \rrbracket \implies P f$
<proof>

lemma *fix-ind2*:
assumes *adm*: $\text{adm } P$
assumes *0*: $P \perp$ **and** *1*: $P (F \cdot \perp)$
assumes *step*: $\bigwedge x. \llbracket P x; P (F \cdot x) \rrbracket \implies P (F \cdot (F \cdot x))$
shows $P (\text{fix} \cdot F)$
<proof>

lemma *parallel-fix-ind*:
assumes *adm*: $\text{adm } (\lambda x. P (\text{fst } x) (\text{snd } x))$
assumes *base*: $P \perp \perp$
assumes *step*: $\bigwedge x y. P x y \implies P (F \cdot x) (G \cdot y)$
shows $P (\text{fix} \cdot F) (\text{fix} \cdot G)$
<proof>

lemma *cont-parallel-fix-ind*:
assumes *cont* *F* **and** *cont* *G*
assumes *adm*: $\text{adm } (\lambda x. P (\text{fst } x) (\text{snd } x))$
assumes $P \perp \perp$
assumes $\bigwedge x y. P x y \implies P (F x) (G y)$
shows $P (\text{fix} \cdot (\text{Abs-cfun } F)) (\text{fix} \cdot (\text{Abs-cfun } G))$
<proof>

25.4 Fixed-points on product types

Bekic’s Theorem: Simultaneous fixed points over pairs can be written in terms of separate fixed points.

lemma *fix-cprod*:

$$\begin{aligned} \text{fix} \cdot (F :: 'a \times 'b \rightarrow 'a \times 'b) = \\ (\mu x. \text{fst} (F \cdot (x, \mu y. \text{snd} (F \cdot (x, y))))), \\ \mu y. \text{snd} (F \cdot (\mu x. \text{fst} (F \cdot (x, \mu y. \text{snd} (F \cdot (x, y))))), y)) \\ (\text{is } \text{fix} \cdot F = (?x, ?y)) \\ \langle \text{proof} \rangle \end{aligned}$$

end

26 Package for defining recursive functions in HOLCF

theory *Fixrec*

imports *Cprod Sprod Ssum Up One Tr Fix*

keywords *fixrec :: thy-defn*

begin

26.1 Pattern-match monad

default-sort *cpo*

pcpodef *'a match = UNIV :: (one ++ 'a u) set*
 $\langle \text{proof} \rangle$

definition

$$\begin{aligned} \text{fail} :: 'a \text{ match } \mathbf{where} \\ \text{fail} = \text{Abs-match} (\text{sinl} \cdot \text{ONE}) \end{aligned}$$

definition

$$\begin{aligned} \text{succeed} :: 'a \rightarrow 'a \text{ match } \mathbf{where} \\ \text{succeed} = (\Lambda x. \text{Abs-match} (\text{sinr} \cdot (\text{up} \cdot x))) \end{aligned}$$

lemma *matchE* [*case-names bottom fail succeed, cases type: match*]:

$$\llbracket p = \perp \implies Q; p = \text{fail} \implies Q; \bigwedge x. p = \text{succeed} \cdot x \implies Q \rrbracket \implies Q$$
 $\langle \text{proof} \rangle$

lemma *succeed-defined* [*simp*]: *succeed* · *x* ≠ ⊥
 $\langle \text{proof} \rangle$

lemma *fail-defined* [*simp*]: *fail* ≠ ⊥
 $\langle \text{proof} \rangle$

lemma *succeed-eq* [*simp*]: (*succeed* · *x* = *succeed* · *y*) = (*x* = *y*)
 $\langle \text{proof} \rangle$

lemma *succeed-neq-fail* [*simp*]:

$succeed \cdot x \neq fail \ fail \neq succeed \cdot x$
 ⟨proof⟩

26.1.1 Run operator

definition

$run :: 'a \ match \rightarrow 'a::pcpo \ \mathbf{where}$
 $run = (\Lambda \ m. \ sscase \cdot \perp \cdot (fup \cdot ID) \cdot (Rep\text{-}match \ m))$

rewrite rules for run

lemma *run-strict* [simp]: $run \cdot \perp = \perp$
 ⟨proof⟩

lemma *run-fail* [simp]: $run \cdot fail = \perp$
 ⟨proof⟩

lemma *run-succeed* [simp]: $run \cdot (succeed \cdot x) = x$
 ⟨proof⟩

26.1.2 Monad plus operator

definition

$mplus :: 'a \ match \rightarrow 'a \ match \rightarrow 'a \ match \ \mathbf{where}$
 $mplus = (\Lambda \ m1 \ m2. \ sscase \cdot (\Lambda \ -. \ m2) \cdot (\Lambda \ -. \ m1) \cdot (Rep\text{-}match \ m1))$

abbreviation

$mplus\text{-}syn :: ['a \ match, 'a \ match] \Rightarrow 'a \ match \ (\mathbf{infixr} \ +++ \ 65) \ \mathbf{where}$
 $m1 \ +++ \ m2 == mplus \cdot m1 \cdot m2$

rewrite rules for mplus

lemma *mplus-strict* [simp]: $\perp \ +++ \ m = \perp$
 ⟨proof⟩

lemma *mplus-fail* [simp]: $fail \ +++ \ m = m$
 ⟨proof⟩

lemma *mplus-succeed* [simp]: $succeed \cdot x \ +++ \ m = succeed \cdot x$
 ⟨proof⟩

lemma *mplus-fail2* [simp]: $m \ +++ \ fail = m$
 ⟨proof⟩

lemma *mplus-assoc*: $(x \ +++ \ y) \ +++ \ z = x \ +++ \ (y \ +++ \ z)$
 ⟨proof⟩

26.2 Match functions for built-in types

default-sort *pcpo*

definition

$$\text{match-bottom} :: 'a \rightarrow 'c \text{ match} \rightarrow 'c \text{ match}$$
where

$$\text{match-bottom} = (\Lambda x k. \text{seq} \cdot x \cdot \text{fail})$$
definition

$$\text{match-Pair} :: 'a :: \text{cpo} \times 'b :: \text{cpo} \rightarrow ('a \rightarrow 'b \rightarrow 'c \text{ match}) \rightarrow 'c \text{ match}$$
where

$$\text{match-Pair} = (\Lambda x k. \text{csplit} \cdot k \cdot x)$$
definition

$$\text{match-spair} :: 'a \otimes 'b \rightarrow ('a \rightarrow 'b \rightarrow 'c \text{ match}) \rightarrow 'c \text{ match}$$
where

$$\text{match-spair} = (\Lambda x k. \text{ssplit} \cdot k \cdot x)$$
definition

$$\text{match-sinl} :: 'a \oplus 'b \rightarrow ('a \rightarrow 'c \text{ match}) \rightarrow 'c \text{ match}$$
where

$$\text{match-sinl} = (\Lambda x k. \text{sscase} \cdot k \cdot (\Lambda b. \text{fail}) \cdot x)$$
definition

$$\text{match-sinr} :: 'a \oplus 'b \rightarrow ('b \rightarrow 'c \text{ match}) \rightarrow 'c \text{ match}$$
where

$$\text{match-sinr} = (\Lambda x k. \text{sscase} \cdot (\Lambda a. \text{fail}) \cdot k \cdot x)$$
definition

$$\text{match-up} :: 'a :: \text{cpo} \ u \rightarrow ('a \rightarrow 'c \text{ match}) \rightarrow 'c \text{ match}$$
where

$$\text{match-up} = (\Lambda x k. \text{fup} \cdot k \cdot x)$$
definition

$$\text{match-ONE} :: \text{one} \rightarrow 'c \text{ match} \rightarrow 'c \text{ match}$$
where

$$\text{match-ONE} = (\Lambda \text{ONE} k. k)$$
definition

$$\text{match-TT} :: \text{tr} \rightarrow 'c \text{ match} \rightarrow 'c \text{ match}$$
where

$$\text{match-TT} = (\Lambda x k. \text{If } x \text{ then } k \text{ else fail})$$
definition

$$\text{match-FF} :: \text{tr} \rightarrow 'c \text{ match} \rightarrow 'c \text{ match}$$
where

$$\text{match-FF} = (\Lambda x k. \text{If } x \text{ then fail else } k)$$
lemma *match-bottom-simps* [*simp*]:
$$\text{match-bottom} \cdot x \cdot k = (\text{if } x = \perp \text{ then } \perp \text{ else fail})$$

(*proof*)

lemma *match-Pair-simps* [simp]:

$$\text{match-Pair} \cdot (x, y) \cdot k = k \cdot x \cdot y$$

<proof>

lemma *match-spair-simps* [simp]:

$$\llbracket x \neq \perp; y \neq \perp \rrbracket \implies \text{match-spair} \cdot (:x, y) \cdot k = k \cdot x \cdot y$$

$$\text{match-spair} \cdot \perp \cdot k = \perp$$

<proof>

lemma *match-sinl-simps* [simp]:

$$x \neq \perp \implies \text{match-sinl} \cdot (\text{sinl} \cdot x) \cdot k = k \cdot x$$

$$y \neq \perp \implies \text{match-sinl} \cdot (\text{sinr} \cdot y) \cdot k = \text{fail}$$

$$\text{match-sinl} \cdot \perp \cdot k = \perp$$

<proof>

lemma *match-sinr-simps* [simp]:

$$x \neq \perp \implies \text{match-sinr} \cdot (\text{sinl} \cdot x) \cdot k = \text{fail}$$

$$y \neq \perp \implies \text{match-sinr} \cdot (\text{sinr} \cdot y) \cdot k = k \cdot y$$

$$\text{match-sinr} \cdot \perp \cdot k = \perp$$

<proof>

lemma *match-up-simps* [simp]:

$$\text{match-up} \cdot (\text{up} \cdot x) \cdot k = k \cdot x$$

$$\text{match-up} \cdot \perp \cdot k = \perp$$

<proof>

lemma *match-ONE-simps* [simp]:

$$\text{match-ONE} \cdot \text{ONE} \cdot k = k$$

$$\text{match-ONE} \cdot \perp \cdot k = \perp$$

<proof>

lemma *match-TT-simps* [simp]:

$$\text{match-TT} \cdot \text{TT} \cdot k = k$$

$$\text{match-TT} \cdot \text{FF} \cdot k = \text{fail}$$

$$\text{match-TT} \cdot \perp \cdot k = \perp$$

<proof>

lemma *match-FF-simps* [simp]:

$$\text{match-FF} \cdot \text{FF} \cdot k = k$$

$$\text{match-FF} \cdot \text{TT} \cdot k = \text{fail}$$

$$\text{match-FF} \cdot \perp \cdot k = \perp$$

<proof>

26.3 Mutual recursion

The following rules are used to prove unfolding theorems from fixed-point definitions of mutually recursive functions.

lemma *Pair-equalI*: $\llbracket x \equiv \text{fst } p; y \equiv \text{snd } p \rrbracket \implies (x, y) \equiv p$

<proof>

lemma *Pair-eqD1*: $(x, y) = (x', y') \implies x = x'$
 ⟨*proof*⟩

lemma *Pair-eqD2*: $(x, y) = (x', y') \implies y = y'$
 ⟨*proof*⟩

lemma *def-cont-fix-eq*:
 $\llbracket f \equiv \text{fix} \cdot (\text{Abs-cfun } F); \text{ cont } F \rrbracket \implies f = F f$
 ⟨*proof*⟩

lemma *def-cont-fix-ind*:
 $\llbracket f \equiv \text{fix} \cdot (\text{Abs-cfun } F); \text{ cont } F; \text{ adm } P; P \perp; \bigwedge x. P x \implies P (F x) \rrbracket \implies P f$
 ⟨*proof*⟩

lemma for proving rewrite rules

lemma *ssubst-lhs*: $\llbracket t = s; P s = Q \rrbracket \implies P t = Q$
 ⟨*proof*⟩

26.4 Initializing the fixrec package

⟨*ML*⟩

hide-const (**open**) *succeed fail run*

end

27 Domain package support

theory *Domain-Aux*
imports *Map-Functions Fixrec*
begin

27.1 Continuous isomorphisms

A locale for continuous isomorphisms

locale *iso* =
fixes *abs* :: 'a → 'b
fixes *rep* :: 'b → 'a
assumes *abs-iso* [*simp*]: *rep* · (*abs* · *x*) = *x*
assumes *rep-iso* [*simp*]: *abs* · (*rep* · *y*) = *y*
begin

lemma *swap*: *iso rep abs*
 ⟨*proof*⟩

lemma *abs-below*: $(\text{abs} \cdot x \sqsubseteq \text{abs} \cdot y) = (x \sqsubseteq y)$
 ⟨*proof*⟩

lemma *rep-below*: $(rep.x \sqsubseteq rep.y) = (x \sqsubseteq y)$
 ⟨proof⟩

lemma *abs-eq*: $(abs.x = abs.y) = (x = y)$
 ⟨proof⟩

lemma *rep-eq*: $(rep.x = rep.y) = (x = y)$
 ⟨proof⟩

lemma *abs-strict*: $abs.\perp = \perp$
 ⟨proof⟩

lemma *rep-strict*: $rep.\perp = \perp$
 ⟨proof⟩

lemma *abs-defin'*: $abs.x = \perp \implies x = \perp$
 ⟨proof⟩

lemma *rep-defin'*: $rep.z = \perp \implies z = \perp$
 ⟨proof⟩

lemma *abs-defined*: $z \neq \perp \implies abs.z \neq \perp$
 ⟨proof⟩

lemma *rep-defined*: $z \neq \perp \implies rep.z \neq \perp$
 ⟨proof⟩

lemma *abs-bottom-iff*: $(abs.x = \perp) = (x = \perp)$
 ⟨proof⟩

lemma *rep-bottom-iff*: $(rep.x = \perp) = (x = \perp)$
 ⟨proof⟩

lemma *casedist-rule*: $rep.x = \perp \vee P \implies x = \perp \vee P$
 ⟨proof⟩

lemma *compact-abs-rev*: $compact (abs.x) \implies compact x$
 ⟨proof⟩

lemma *compact-rep-rev*: $compact (rep.x) \implies compact x$
 ⟨proof⟩

lemma *compact-abs*: $compact x \implies compact (abs.x)$
 ⟨proof⟩

lemma *compact-rep*: $compact x \implies compact (rep.x)$
 ⟨proof⟩

lemma *iso-swap*: $(x = \text{abs}\cdot y) = (\text{rep}\cdot x = y)$
 ⟨*proof*⟩

end

27.2 Proofs about take functions

This section contains lemmas that are used in a module that supports the domain isomorphism package; the module contains proofs related to take functions and the finiteness predicate.

lemma *deflation-abs-rep*:
fixes *abs* **and** *rep* **and** *d*
assumes *abs-iso*: $\bigwedge x. \text{rep}\cdot(\text{abs}\cdot x) = x$
assumes *rep-iso*: $\bigwedge y. \text{abs}\cdot(\text{rep}\cdot y) = y$
shows *deflation* *d* \implies *deflation* (*abs* *oo* *d* *oo* *rep*)
 ⟨*proof*⟩

lemma *deflation-chain-min*:
assumes *chain*: *chain* *d*
assumes *defl*: $\bigwedge n. \text{deflation}$ (*d* *n*)
shows *d* *m*·(*d* *n*·*x*) = *d* (*min* *m* *n*)·*x*
 ⟨*proof*⟩

lemma *lub-ID-take-lemma*:
assumes *chain* *t* **and** $(\bigsqcup n. t\ n) = \text{ID}$
assumes $\bigwedge n. t\ n\cdot x = t\ n\cdot y$ **shows** $x = y$
 ⟨*proof*⟩

lemma *lub-ID-reach*:
assumes *chain* *t* **and** $(\bigsqcup n. t\ n) = \text{ID}$
shows $(\bigsqcup n. t\ n\cdot x) = x$
 ⟨*proof*⟩

lemma *lub-ID-take-induct*:
assumes *chain* *t* **and** $(\bigsqcup n. t\ n) = \text{ID}$
assumes *adm* *P* **and** $\bigwedge n. P$ (*t* *n*·*x*) **shows** *P* *x*
 ⟨*proof*⟩

27.3 Finiteness

Let a “decisive” function be a deflation that maps every input to either itself or bottom. Then if a domain’s take functions are all decisive, then all values in the domain are finite.

definition

decisive :: $(\text{'a}::\text{pcpo} \rightarrow \text{'a}) \Rightarrow \text{bool}$

where

decisive *d* $\iff (\forall x. d\cdot x = x \vee d\cdot x = \perp)$

lemma *decisiveI*: $(\bigwedge x. d \cdot x = x \vee d \cdot x = \perp) \implies \text{decisive } d$
 ⟨proof⟩

lemma *decisive-cases*:
assumes *decisive d* **obtains** $d \cdot x = x \mid d \cdot x = \perp$
 ⟨proof⟩

lemma *decisive-bottom*: *decisive* \perp
 ⟨proof⟩

lemma *decisive-ID*: *decisive* *ID*
 ⟨proof⟩

lemma *decisive-ssum-map*:
assumes *f*: *decisive f*
assumes *g*: *decisive g*
shows *decisive* (*ssum-map.f.g*)
 ⟨proof⟩

lemma *decisive-sprod-map*:
assumes *f*: *decisive f*
assumes *g*: *decisive g*
shows *decisive* (*sprod-map.f.g*)
 ⟨proof⟩

lemma *decisive-abs-rep*:
fixes *abs rep*
assumes *iso*: *iso abs rep*
assumes *d*: *decisive d*
shows *decisive* (*abs oo d oo rep*)
 ⟨proof⟩

lemma *lub-ID-finite*:
assumes *chain*: *chain d*
assumes *lub*: $(\bigsqcup n. d \ n) = ID$
assumes *decisive*: $\bigwedge n. \text{decisive } (d \ n)$
shows $\exists n. d \ n \cdot x = x$
 ⟨proof⟩

lemma *lub-ID-finite-take-induct*:
assumes *chain d* **and** $(\bigsqcup n. d \ n) = ID$ **and** $\bigwedge n. \text{decisive } (d \ n)$
shows $(\bigwedge n. P \ (d \ n \cdot x)) \implies P \ x$
 ⟨proof⟩

27.4 Proofs about constructor functions

Lemmas for proving nchotomy rule:

lemma *ex-one-bottom-iff*:
 $(\exists x. P \ x \wedge x \neq \perp) = P \ ONE$

<proof>

lemma *ex-up-bottom-iff*:

$$(\exists x. P x \wedge x \neq \perp) = (\exists x. P (up \cdot x))$$

<proof>

lemma *ex-sprod-bottom-iff*:

$$\begin{aligned} (\exists y. P y \wedge y \neq \perp) = \\ (\exists x y. (P (:x, y) \wedge x \neq \perp) \wedge y \neq \perp) \end{aligned}$$

<proof>

lemma *ex-sprod-up-bottom-iff*:

$$\begin{aligned} (\exists y. P y \wedge y \neq \perp) = \\ (\exists x y. P (:up \cdot x, y) \wedge y \neq \perp) \end{aligned}$$

<proof>

lemma *ex-ssum-bottom-iff*:

$$\begin{aligned} (\exists x. P x \wedge x \neq \perp) = \\ ((\exists x. P (sinl \cdot x) \wedge x \neq \perp) \vee \\ (\exists x. P (sinr \cdot x) \wedge x \neq \perp)) \end{aligned}$$

<proof>

lemma *exh-start*: $p = \perp \vee (\exists x. p = x \wedge x \neq \perp)$

<proof>

lemmas *ex-bottom-iffs* =

ex-ssum-bottom-iff
ex-sprod-up-bottom-iff
ex-sprod-bottom-iff
ex-up-bottom-iff
ex-one-bottom-iff

Rules for turning nchotomy into exhaust:

lemma *exh-casedist0*: $\llbracket R; R \implies P \rrbracket \implies P$

<proof>

lemma *exh-casedist1*: $((P \vee Q \implies R) \implies S) \equiv (\llbracket P \implies R; Q \implies R \rrbracket \implies S)$

<proof>

lemma *exh-casedist2*: $(\exists x. P x \implies Q) \equiv (\wedge x. P x \implies Q)$

<proof>

lemma *exh-casedist3*: $(P \wedge Q \implies R) \equiv (P \implies Q \implies R)$

<proof>

lemmas *exh-casedists* = *exh-casedist1* *exh-casedist2* *exh-casedist3*

Rules for proving constructor properties

lemmas *con-strict-rules* =

sinl-strict sinr-strict spair-strict1 spair-strict2

lemmas *con-bottom-iff-rules =*
sinl-bottom-iff sinr-bottom-iff spair-bottom-iff up-defined ONE-defined

lemmas *con-below-iff-rules =*
sinl-below sinr-below sinl-below-sinr sinr-below-sinl con-bottom-iff-rules

lemmas *con-eq-iff-rules =*
sinl-eq sinr-eq sinl-eq-sinr sinr-eq-sinl con-bottom-iff-rules

lemmas *sel-strict-rules =*
cfcomp2 sscase1 sfst-strict ssnd-strict fup1

lemma *sel-app-extra-rules:*

sscase.ID.⊥.(sinr.x) = ⊥
sscase.ID.⊥.(sinl.x) = x
sscase.⊥.ID.(sinl.x) = ⊥
sscase.⊥.ID.(sinr.x) = x
fup.ID.(up.x) = x

<proof>

lemmas *sel-app-rules =*
sel-strict-rules sel-app-extra-rules
ssnd-spair sfst-spair up-defined spair-defined

lemmas *sel-bottom-iff-rules =*
cfcomp2 sfst-bottom-iff ssnd-bottom-iff

lemmas *take-con-rules =*
ssum-map-sinl' ssum-map-sinr' sprod-map-spair' u-map-up
deflation-strict deflation-ID ID1 cfcomp2

27.5 ML setup

named-theorems *domain-deflation theorems like deflation a ==> deflation (foo-map\$a)*
and *domain-map-ID theorems like foo-map\$ID = ID*

<ML>

end

28 Domain package

theory *Domain*
imports *Representable Domain-Aux*
keywords
lazy unsafe and
domaindef domain :: thy-defn and

```

domain-isomorphism :: thy-decl
begin

```

```

default-sort domain

```

28.1 Representations of types

```

lemma emb-prj: emb·((prj·x)::'a) = cast·DEFL('a)·x
⟨proof⟩

```

```

lemma emb-prj-emb:
  fixes x :: 'a
  assumes DEFL('a) ⊆ DEFL('b)
  shows emb·(prj·(emb·x) :: 'b) = emb·x
⟨proof⟩

```

```

lemma prj-emb-prj:
  assumes DEFL('a) ⊆ DEFL('b)
  shows prj·(emb·(prj·x :: 'b)) = (prj·x :: 'a)
⟨proof⟩

```

Isomorphism lemmas used internally by the domain package:

```

lemma domain-abs-iso:
  fixes abs and rep
  assumes DEFL: DEFL('b) = DEFL('a)
  assumes abs-def: (abs :: 'a → 'b) ≡ prj oo emb
  assumes rep-def: (rep :: 'b → 'a) ≡ prj oo emb
  shows rep·(abs·x) = x
⟨proof⟩

```

```

lemma domain-rep-iso:
  fixes abs and rep
  assumes DEFL: DEFL('b) = DEFL('a)
  assumes abs-def: (abs :: 'a → 'b) ≡ prj oo emb
  assumes rep-def: (rep :: 'b → 'a) ≡ prj oo emb
  shows abs·(rep·x) = x
⟨proof⟩

```

28.2 Deflations as sets

```

definition defl-set :: 'a::bifinite defl ⇒ 'a set
where defl-set A = {x. cast·A·x = x}

```

```

lemma adm-defl-set: adm (λx. x ∈ defl-set A)
⟨proof⟩

```

```

lemma defl-set-bottom: ⊥ ∈ defl-set A
⟨proof⟩

```

lemma *defl-set-cast* [*simp*]: $\text{cast} \cdot A \cdot x \in \text{defl-set } A$
 ⟨*proof*⟩

lemma *defl-set-subset-iff*: $\text{defl-set } A \subseteq \text{defl-set } B \iff A \sqsubseteq B$
 ⟨*proof*⟩

28.3 Proving a subtype is representable

Temporarily relax type constraints.

⟨*ML*⟩

lemma *typedef-domain-class*:
 fixes *Rep* :: 'a::pcpo \Rightarrow *udom*
 fixes *Abs* :: *udom* \Rightarrow 'a::pcpo
 fixes *t* :: *udom* *defl*
 assumes *type-definition* *Rep* *Abs* (*defl-set t*)
 assumes *below*: $(\sqsubseteq) \equiv \lambda x y. \text{Rep } x \sqsubseteq \text{Rep } y$
 assumes *emb*: $\text{emb} \equiv (\Lambda x. \text{Rep } x)$
 assumes *prj*: $\text{prj} \equiv (\Lambda x. \text{Abs } (\text{cast} \cdot t \cdot x))$
 assumes *defl*: $\text{defl} \equiv (\lambda a::'a \text{ itself}. t)$
 assumes *liftemb*: $(\text{liftemb} :: 'a \ u \rightarrow \text{udom } u) \equiv u\text{-map} \cdot \text{emb}$
 assumes *liftprj*: $(\text{liftprj} :: \text{udom } u \rightarrow 'a \ u) \equiv u\text{-map} \cdot \text{prj}$
 assumes *liftdefl*: $(\text{liftdefl} :: 'a \ \text{itself} \Rightarrow -) \equiv (\lambda t. \text{liftdefl-of} \cdot \text{DEFL}('a))$
 shows *OFCLASS*('a, *domain-class*)
 ⟨*proof*⟩

lemma *typedef-DEFL*:
 assumes *defl* $\equiv (\lambda a::'a::\text{pcpo} \ \text{itself}. t)$
 shows *DEFL*('a::pcpo) = *t*
 ⟨*proof*⟩

Restore original typing constraints.

⟨*ML*⟩

28.4 Isomorphic deflations

definition *isodefl* :: ('a \rightarrow 'a) \Rightarrow *udom* *defl* \Rightarrow *bool*
 where *isodefl* *d t* $\iff \text{cast} \cdot t = \text{emb} \ \text{oo} \ d \ \text{oo} \ \text{prj}$

definition *isodefl'* :: ('a::predomain \rightarrow 'a) \Rightarrow *udom* *u* *defl* \Rightarrow *bool*
 where *isodefl'* *d t* $\iff \text{cast} \cdot t = \text{liftemb} \ \text{oo} \ u\text{-map} \cdot d \ \text{oo} \ \text{liftprj}$

lemma *isodeflI*: $(\bigwedge x. \text{cast} \cdot t \cdot x = \text{emb} \cdot (d \cdot (\text{prj} \cdot x))) \implies \text{isodefl } d \ t$
 ⟨*proof*⟩

lemma *cast-isodefl*: $\text{isodefl } d \ t \implies \text{cast} \cdot t = (\Lambda x. \text{emb} \cdot (d \cdot (\text{prj} \cdot x)))$
 ⟨*proof*⟩

lemma *isodefl-strict*: $\text{isodefl } d \ t \implies d \cdot \perp = \perp$

<proof>

lemma *isodefl-imp-deflation*:

fixes $d :: 'a \rightarrow 'a$

assumes *isodefl* d **shows** *deflation* d

<proof>

lemma *isodefl-ID-DEFL*: *isodefl* ($ID :: 'a \rightarrow 'a$) *DEFL*('a)

<proof>

lemma *isodefl-LIFTDEFL*:

isodefl' ($ID :: 'a \rightarrow 'a$) *LIFTDEFL*('a::predomain)

<proof>

lemma *isodefl-DEFL-imp-ID*: *isodefl* ($d :: 'a \rightarrow 'a$) *DEFL*('a) $\implies d = ID$

<proof>

lemma *isodefl-bottom*: *isodefl* $\perp \perp$

<proof>

lemma *adm-isodefl*:

cont $f \implies \text{cont } g \implies \text{adm } (\lambda x. \text{isodefl } (f x) (g x))$

<proof>

lemma *isodefl-lub*:

assumes *chain* d **and** *chain* t

assumes $\bigwedge i. \text{isodefl } (d i) (t i)$

shows *isodefl* ($\bigsqcup i. d i$) ($\bigsqcup i. t i$)

<proof>

lemma *isodefl-fix*:

assumes $\bigwedge d t. \text{isodefl } d t \implies \text{isodefl } (f \cdot d) (g \cdot t)$

shows *isodefl* ($\text{fix} \cdot f$) ($\text{fix} \cdot g$)

<proof>

lemma *isodefl-abs-rep*:

fixes *abs* **and** *rep* **and** d

assumes *DEFL*: *DEFL*('b) = *DEFL*('a)

assumes *abs-def*: ($\text{abs} :: 'a \rightarrow 'b$) $\equiv \text{prj} \circ \text{emb}$

assumes *rep-def*: ($\text{rep} :: 'b \rightarrow 'a$) $\equiv \text{prj} \circ \text{emb}$

shows *isodefl* $d t \implies \text{isodefl } (\text{abs} \circ d \circ \text{rep}) t$

<proof>

lemma *isodefl'-liftdefl-of*: *isodefl* $d t \implies \text{isodefl}' d (\text{liftdefl-of} \cdot t)$

<proof>

lemma *isodefl-sfun*:

isodefl $d_1 t_1 \implies \text{isodefl } d_2 t_2 \implies$

isodefl ($\text{sfun-map} \cdot d_1 \cdot d_2$) ($\text{sfun-defl} \cdot t_1 \cdot t_2$)

<proof>

lemma *isodefl-ssum*:

$isodefl\ d1\ t1 \implies isodefl\ d2\ t2 \implies$
 $isodefl\ (ssum\ map\cdot d1\cdot d2)\ (ssum\ defl\cdot t1\cdot t2)$

<proof>

lemma *isodefl-sprod*:

$isodefl\ d1\ t1 \implies isodefl\ d2\ t2 \implies$
 $isodefl\ (sprod\ map\cdot d1\cdot d2)\ (sprod\ defl\cdot t1\cdot t2)$

<proof>

lemma *isodefl-prod*:

$isodefl\ d1\ t1 \implies isodefl\ d2\ t2 \implies$
 $isodefl\ (prod\ map\cdot d1\cdot d2)\ (prod\ defl\cdot t1\cdot t2)$

<proof>

lemma *isodefl-u*:

$isodefl\ d\ t \implies isodefl\ (u\ map\cdot d)\ (u\ defl\cdot t)$

<proof>

lemma *isodefl-u-liftdefl*:

$isodefl'\ d\ t \implies isodefl\ (u\ map\cdot d)\ (u\ liftdefl\cdot t)$

<proof>

lemma *encode-prod-u-map*:

$encode\ prod\ u\cdot (u\ map\cdot (prod\ map\cdot f\cdot g)\cdot (decode\ prod\ u\cdot x))$
 $=\ sprod\ map\cdot (u\ map\cdot f)\cdot (u\ map\cdot g)\cdot x$

<proof>

lemma *isodefl-prod-u*:

assumes $isodefl'\ d1\ t1$ **and** $isodefl'\ d2\ t2$
shows $isodefl'\ (prod\ map\cdot d1\cdot d2)\ (prod\ liftdefl\cdot t1\cdot t2)$

<proof>

lemma *encode-cfun-map*:

$encode\ cfun\cdot (cfun\ map\cdot f\cdot g\cdot (decode\ cfun\cdot x))$
 $=\ sfun\ map\cdot (u\ map\cdot f)\cdot g\cdot x$

<proof>

lemma *isodefl-cfun*:

assumes $isodefl\ (u\ map\cdot d1)\ t1$ **and** $isodefl\ d2\ t2$
shows $isodefl\ (cfun\ map\cdot d1\cdot d2)\ (sfun\ defl\cdot t1\cdot t2)$

<proof>

28.5 Setting up the domain package

named-theorems *domain-defl-simps* theorems like $DEFL('a\ t) = t\ defl\ \$\ DEFL('a)$
and *domain-isodefl* theorems like $isodefl\ d\ t \implies isodefl\ (foo\ map\ \$\ d)\ (foo\ defl\ \$\ t)$

⟨ML⟩

lemmas [domain-defl-simps] =
DEFL-cfun DEFL-sfun DEFL-ssum DEFL-sprod DEFL-prod DEFL-u
liftdefl-eq LIFTDEFL-prod u-liftdefl-liftdefl-of

lemmas [domain-map-ID] =
cfun-map-ID sfun-map-ID ssum-map-ID sprod-map-ID prod-map-ID u-map-ID

lemmas [domain-isodefl] =
isodefl-u isodefl-sfun isodefl-ssum isodefl-sprod
isodefl-cfun isodefl-prod isodefl-prod-u isodefl'-liftdefl-of
isodefl-u-liftdefl

lemmas [domain-deflation] =
deflation-cfun-map deflation-sfun-map deflation-ssum-map
deflation-sprod-map deflation-prod-map deflation-u-map

⟨ML⟩

end

29 A compact basis for powerdomains

theory *Compact-Basis*

imports *Universal*

begin

default-sort *bifinite*

29.1 A compact basis for powerdomains

definition *pd-basis* = {*S*::'a compact-basis set. finite *S* ∧ *S* ≠ {}}

typedef 'a *pd-basis* = *pd-basis* :: 'a compact-basis set set
 ⟨proof⟩

lemma *finite-Rep-pd-basis* [simp]: finite (Rep-*pd-basis* u)
 ⟨proof⟩

lemma *Rep-pd-basis-nonempty* [simp]: Rep-*pd-basis* u ≠ {}
 ⟨proof⟩

The powerdomain basis type is countable.

lemma *pd-basis-countable*: ∃ *f*::'a *pd-basis* ⇒ nat. inj *f*
 ⟨proof⟩

29.2 Unit and plus constructors

definition

$PDUnit :: 'a \text{ compact-basis} \Rightarrow 'a \text{ pd-basis}$ **where**
 $PDUnit = (\lambda x. \text{Abs-pd-basis } \{x\})$

definition

$PDPlus :: 'a \text{ pd-basis} \Rightarrow 'a \text{ pd-basis} \Rightarrow 'a \text{ pd-basis}$ **where**
 $PDPlus \ t \ u = \text{Abs-pd-basis } (\text{Rep-pd-basis } t \cup \text{Rep-pd-basis } u)$

lemma *Rep-PDUnit*:

$\text{Rep-pd-basis } (PDUnit \ x) = \{x\}$
 $\langle \text{proof} \rangle$

lemma *Rep-PDPlus*:

$\text{Rep-pd-basis } (PDPlus \ u \ v) = \text{Rep-pd-basis } u \cup \text{Rep-pd-basis } v$
 $\langle \text{proof} \rangle$

lemma *PDUnit-inject* [*simp*]: $(PDUnit \ a = PDUnit \ b) = (a = b)$

$\langle \text{proof} \rangle$

lemma *PDPlus-assoc*: $PDPlus \ (PDPlus \ t \ u) \ v = PDPlus \ t \ (PDPlus \ u \ v)$

$\langle \text{proof} \rangle$

lemma *PDPlus-commute*: $PDPlus \ t \ u = PDPlus \ u \ t$

$\langle \text{proof} \rangle$

lemma *PDPlus-absorb*: $PDPlus \ t \ t = t$

$\langle \text{proof} \rangle$

lemma *pd-basis-induct1*:

assumes $PDUnit: \bigwedge a. P \ (PDUnit \ a)$
assumes $PDPlus: \bigwedge a \ t. P \ t \Longrightarrow P \ (PDPlus \ (PDUnit \ a) \ t)$
shows $P \ x$

$\langle \text{proof} \rangle$

lemma *pd-basis-induct*:

assumes $PDUnit: \bigwedge a. P \ (PDUnit \ a)$
assumes $PDPlus: \bigwedge t \ u. \llbracket P \ t; P \ u \rrbracket \Longrightarrow P \ (PDPlus \ t \ u)$
shows $P \ x$

$\langle \text{proof} \rangle$

29.3 Fold operator

definition

$fold\text{-pd} ::$
 $('a \text{ compact-basis} \Rightarrow 'b::\text{type}) \Rightarrow ('b \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'a \text{ pd-basis} \Rightarrow 'b$
where $fold\text{-pd} \ g \ f \ t = \text{semilattice-set.F } f \ (g \ ' \ \text{Rep-pd-basis } t)$

lemma *fold-pd-PDUnit*:

assumes *semilattice f*
shows $\text{fold-pd } g \ f \ (\text{PDUnit } x) = g \ x$
 $\langle \text{proof} \rangle$

lemma *fold-pd-PDPlus:*
assumes *semilattice f*
shows $\text{fold-pd } g \ f \ (\text{PDPlus } t \ u) = f \ (\text{fold-pd } g \ f \ t) \ (\text{fold-pd } g \ f \ u)$
 $\langle \text{proof} \rangle$

end

30 Upper powerdomain

theory *UpperPD*
imports *Compact-Basis*
begin

30.1 Basis preorder

definition
 $\text{upper-le} :: 'a \ \text{pd-basis} \Rightarrow 'a \ \text{pd-basis} \Rightarrow \text{bool}$ (**infix** $\leq\#$ 50) **where**
 $\text{upper-le} = (\lambda u \ v. \forall y \in \text{Rep-pd-basis } v. \exists x \in \text{Rep-pd-basis } u. x \sqsubseteq y)$

lemma *upper-le-refl* [*simp*]: $t \leq\# t$
 $\langle \text{proof} \rangle$

lemma *upper-le-trans*: $\llbracket t \leq\# u; u \leq\# v \rrbracket \Longrightarrow t \leq\# v$
 $\langle \text{proof} \rangle$

interpretation *upper-le*: *preorder upper-le*
 $\langle \text{proof} \rangle$

lemma *upper-le-minimal* [*simp*]: $\text{PDUnit } \text{compact-bot} \leq\# t$
 $\langle \text{proof} \rangle$

lemma *PDUnit-upper-mono*: $x \sqsubseteq y \Longrightarrow \text{PDUnit } x \leq\# \text{PDUnit } y$
 $\langle \text{proof} \rangle$

lemma *PDPlus-upper-mono*: $\llbracket s \leq\# t; u \leq\# v \rrbracket \Longrightarrow \text{PDPlus } s \ u \leq\# \text{PDPlus } t \ v$
 $\langle \text{proof} \rangle$

lemma *PDPlus-upper-le*: $\text{PDPlus } t \ u \leq\# t$
 $\langle \text{proof} \rangle$

lemma *upper-le-PDUnit-PDUnit-iff* [*simp*]:
 $(\text{PDUnit } a \leq\# \text{PDUnit } b) = (a \sqsubseteq b)$
 $\langle \text{proof} \rangle$

lemma *upper-le-PDPlus-PDUnit-iff*:

$(PDPlus\ t\ u\ \leq\# \ PDUnit\ a) = (t\ \leq\# \ PDUnit\ a \vee u\ \leq\# \ PDUnit\ a)$
 $\langle proof \rangle$

lemma *upper-le-PDPlus-iff*: $(t\ \leq\# \ PDPlus\ u\ v) = (t\ \leq\# \ u \wedge t\ \leq\# \ v)$
 $\langle proof \rangle$

lemma *upper-le-induct* [*induct set: upper-le*]:

assumes *le*: $t\ \leq\# \ u$

assumes 1: $\bigwedge a\ b. a\ \sqsubseteq\ b \implies P\ (PDUnit\ a)\ (PDUnit\ b)$

assumes 2: $\bigwedge t\ u\ a. P\ t\ (PDUnit\ a) \implies P\ (PDPlus\ t\ u)\ (PDUnit\ a)$

assumes 3: $\bigwedge t\ u\ v. \llbracket P\ t\ u; P\ t\ v \rrbracket \implies P\ t\ (PDPlus\ u\ v)$

shows $P\ t\ u$

$\langle proof \rangle$

30.2 Type definition

typedef *'a upper-pd* $(((-)\#)) =$
 $\{S :: 'a\ pd\ basis\ set.\ upper\ le.\ ideal\ S\}$
 $\langle proof \rangle$

instantiation *upper-pd* :: (*bifinite*) *below*
begin

definition

$x\ \sqsubseteq\ y \longleftrightarrow Rep\ upper\ pd\ x\ \subseteq\ Rep\ upper\ pd\ y$

instance $\langle proof \rangle$

end

instance *upper-pd* :: (*bifinite*) *po*
 $\langle proof \rangle$

instance *upper-pd* :: (*bifinite*) *cpo*
 $\langle proof \rangle$

definition

upper-principal :: *'a pd-basis* \Rightarrow *'a upper-pd* **where**

upper-principal $t = Abs\ upper\ pd\ \{u. u\ \leq\# \ t\}$

interpretation *upper-pd*:

ideal-completion upper-le upper-principal Rep-upper-pd

$\langle proof \rangle$

Upper powerdomain is pointed

lemma *upper-pd-minimal*: *upper-principal* (*PDUnit compact-bot*) $\sqsubseteq\ ys$
 $\langle proof \rangle$

instance *upper-pd* :: (*bifinite*) *pcpo*
 $\langle proof \rangle$

lemma *inst-upper-pd-pcpo*: $\perp = \text{upper-principal } (PDUnit \text{ compact-bot})$
 ⟨proof⟩

30.3 Monadic unit and plus

definition

upper-unit :: 'a → 'a *upper-pd* **where**
upper-unit = *compact-basis.extension* ($\lambda a. \text{upper-principal } (PDUnit a)$)

definition

upper-plus :: 'a *upper-pd* → 'a *upper-pd* → 'a *upper-pd* **where**
upper-plus = *upper-pd.extension* ($\lambda t. \text{upper-pd.extension } (\lambda u. \text{upper-principal } (PDPlus t u))$)

abbreviation

upper-add :: 'a *upper-pd* ⇒ 'a *upper-pd* ⇒ 'a *upper-pd*
 (**infixl** $\cup\#$ 65) **where**
 $xs \cup\# ys == \text{upper-plus} \cdot xs \cdot ys$

syntax

-upper-pd :: args ⇒ logic ($\{-\}\#$)

translations

$\{x, xs\}\# == \{x\}\# \cup\# \{xs\}\#$
 $\{x\}\# == \text{CONST } \text{upper-unit} \cdot x$

lemma *upper-unit-Rep-compact-basis* [*simp*]:
 $\{\text{Rep-compact-basis } a\}\# = \text{upper-principal } (PDUnit a)$
 ⟨proof⟩

lemma *upper-plus-principal* [*simp*]:
 $\text{upper-principal } t \cup\# \text{upper-principal } u = \text{upper-principal } (PDPlus t u)$
 ⟨proof⟩

interpretation *upper-add*: *semilattice upper-add* ⟨proof⟩

lemmas *upper-plus-assoc* = *upper-add.assoc*

lemmas *upper-plus-commute* = *upper-add.commute*

lemmas *upper-plus-absorb* = *upper-add.idem*

lemmas *upper-plus-left-commute* = *upper-add.left-commute*

lemmas *upper-plus-left-absorb* = *upper-add.left-idem*

Useful for *simp add*: *upper-plus-ac*

lemmas *upper-plus-ac* =
upper-plus-assoc upper-plus-commute upper-plus-left-commute

Useful for *simp only*: *upper-plus-aci*

lemmas *upper-plus-aci* =

upper-plus-ac upper-plus-absorb upper-plus-left-absorb

lemma *upper-plus-below1*: $xs \cup\# ys \sqsubseteq xs$
 ⟨proof⟩

lemma *upper-plus-below2*: $xs \cup\# ys \sqsubseteq ys$
 ⟨proof⟩

lemma *upper-plus-greatest*: $\llbracket xs \sqsubseteq ys; xs \sqsubseteq zs \rrbracket \implies xs \sqsubseteq ys \cup\# zs$
 ⟨proof⟩

lemma *upper-below-plus-iff* [simp]:
 $xs \sqsubseteq ys \cup\# zs \longleftrightarrow xs \sqsubseteq ys \wedge xs \sqsubseteq zs$
 ⟨proof⟩

lemma *upper-plus-below-unit-iff* [simp]:
 $xs \cup\# ys \sqsubseteq \{z\}\# \longleftrightarrow xs \sqsubseteq \{z\}\# \vee ys \sqsubseteq \{z\}\#$
 ⟨proof⟩

lemma *upper-unit-below-iff* [simp]: $\{x\}\# \sqsubseteq \{y\}\# \longleftrightarrow x \sqsubseteq y$
 ⟨proof⟩

lemmas *upper-pd-below-simps* =
upper-unit-below-iff
upper-below-plus-iff
upper-plus-below-unit-iff

lemma *upper-unit-eq-iff* [simp]: $\{x\}\# = \{y\}\# \longleftrightarrow x = y$
 ⟨proof⟩

lemma *upper-unit-strict* [simp]: $\{\perp\}\# = \perp$
 ⟨proof⟩

lemma *upper-plus-strict1* [simp]: $\perp \cup\# ys = \perp$
 ⟨proof⟩

lemma *upper-plus-strict2* [simp]: $xs \cup\# \perp = \perp$
 ⟨proof⟩

lemma *upper-unit-bottom-iff* [simp]: $\{x\}\# = \perp \longleftrightarrow x = \perp$
 ⟨proof⟩

lemma *upper-plus-bottom-iff* [simp]:
 $xs \cup\# ys = \perp \longleftrightarrow xs = \perp \vee ys = \perp$
 ⟨proof⟩

lemma *compact-upper-unit*: *compact* $x \implies$ *compact* $\{x\}\#$
 ⟨proof⟩

lemma *compact-upper-unit-iff* [simp]: $\text{compact } \{x\}\# \longleftrightarrow \text{compact } x$
 ⟨proof⟩

lemma *compact-upper-plus* [simp]:
 $\llbracket \text{compact } xs; \text{compact } ys \rrbracket \implies \text{compact } (xs \cup\# ys)$
 ⟨proof⟩

30.4 Induction rules

lemma *upper-pd-induct1*:
 assumes $P: \text{adm } P$
 assumes *unit*: $\bigwedge x. P \{x\}\#$
 assumes *insert*: $\bigwedge x ys. \llbracket P \{x\}\#; P ys \rrbracket \implies P (\{x\}\# \cup\# ys)$
 shows $P (xs::'a \text{ upper-pd})$
 ⟨proof⟩

lemma *upper-pd-induct*
 [case-names adm upper-unit upper-plus, induct type: upper-pd]:
 assumes $P: \text{adm } P$
 assumes *unit*: $\bigwedge x. P \{x\}\#$
 assumes *plus*: $\bigwedge xs ys. \llbracket P xs; P ys \rrbracket \implies P (xs \cup\# ys)$
 shows $P (xs::'a \text{ upper-pd})$
 ⟨proof⟩

30.5 Monadic bind

definition

upper-bind-basis ::
 $'a \text{ pd-basis} \Rightarrow ('a \rightarrow 'b \text{ upper-pd}) \rightarrow 'b \text{ upper-pd}$ **where**
 $\text{upper-bind-basis} = \text{fold-pd}$
 $(\lambda a. \Lambda f. f \cdot (\text{Rep-compact-basis } a))$
 $(\lambda x y. \Lambda f. x \cdot f \cup\# y \cdot f)$

lemma *ACI-upper-bind*:
 $\text{semilattice } (\lambda x y. \Lambda f. x \cdot f \cup\# y \cdot f)$
 ⟨proof⟩

lemma *upper-bind-basis-simps* [simp]:
 $\text{upper-bind-basis } (\text{PDUUnit } a) =$
 $(\Lambda f. f \cdot (\text{Rep-compact-basis } a))$
 $\text{upper-bind-basis } (\text{PDPlus } t u) =$
 $(\Lambda f. \text{upper-bind-basis } t \cdot f \cup\# \text{upper-bind-basis } u \cdot f)$
 ⟨proof⟩

lemma *upper-bind-basis-mono*:
 $t \leq\# u \implies \text{upper-bind-basis } t \sqsubseteq \text{upper-bind-basis } u$
 ⟨proof⟩

definition

upper-bind :: $'a \text{ upper-pd} \rightarrow ('a \rightarrow 'b \text{ upper-pd}) \rightarrow 'b \text{ upper-pd}$ **where**

$upper-bind = upper-pd.extension\ upper-bind-basis$

syntax

$-upper-bind :: [logic, logic, logic] \Rightarrow logic$
 $((\exists \cup \# \in \cdot / \cdot) [0, 0, 10] 10)$

translations

$\cup \# x \in xs. e == CONST\ upper-bind \cdot xs \cdot (\Lambda x. e)$

lemma *upper-bind-principal* [simp]:

$upper-bind \cdot (upper-principal\ t) = upper-bind-basis\ t$
 $\langle proof \rangle$

lemma *upper-bind-unit* [simp]:

$upper-bind \cdot \{x\} \# \cdot f = f \cdot x$
 $\langle proof \rangle$

lemma *upper-bind-plus* [simp]:

$upper-bind \cdot (xs \cup \# ys) \cdot f = upper-bind \cdot xs \cdot f \cup \# upper-bind \cdot ys \cdot f$
 $\langle proof \rangle$

lemma *upper-bind-strict* [simp]: $upper-bind \cdot \perp \cdot f = f \cdot \perp$

$\langle proof \rangle$

lemma *upper-bind-bind*:

$upper-bind \cdot (upper-bind \cdot xs \cdot f) \cdot g = upper-bind \cdot xs \cdot (\Lambda x. upper-bind \cdot (f \cdot x) \cdot g)$
 $\langle proof \rangle$

30.6 Map

definition

$upper-map :: ('a \rightarrow 'b) \rightarrow 'a\ upper-pd \rightarrow 'b\ upper-pd$ **where**
 $upper-map = (\Lambda f\ xs. upper-bind \cdot xs \cdot (\Lambda x. \{f \cdot x\} \#))$

lemma *upper-map-unit* [simp]:

$upper-map \cdot f \cdot \{x\} \# = \{f \cdot x\} \#$
 $\langle proof \rangle$

lemma *upper-map-plus* [simp]:

$upper-map \cdot f \cdot (xs \cup \# ys) = upper-map \cdot f \cdot xs \cup \# upper-map \cdot f \cdot ys$
 $\langle proof \rangle$

lemma *upper-map-bottom* [simp]: $upper-map \cdot f \cdot \perp = \{f \cdot \perp\} \#$

$\langle proof \rangle$

lemma *upper-map-ident*: $upper-map \cdot (\Lambda x. x) \cdot xs = xs$

$\langle proof \rangle$

lemma *upper-map-ID*: $upper-map \cdot ID = ID$

<proof>

lemma *upper-map-map*:

$upper-map.f.(upper-map.g.xs) = upper-map.(\Lambda x. f.(g.x)).xs$
<proof>

lemma *upper-bind-map*:

$upper-bind.(upper-map.f.xs).g = upper-bind.xs.(\Lambda x. g.(f.x))$
<proof>

lemma *upper-map-bind*:

$upper-map.f.(upper-bind.xs.g) = upper-bind.xs.(\Lambda x. upper-map.f.(g.x))$
<proof>

lemma *ep-pair-upper-map*: $ep-pair\ e\ p \implies ep-pair\ (upper-map.e)\ (upper-map.p)$

<proof>

lemma *deflation-upper-map*: $deflation\ d \implies deflation\ (upper-map.d)$

<proof>

lemma *finite-deflation-upper-map*:

assumes *finite-deflation* d **shows** *finite-deflation* $(upper-map.d)$
<proof>

30.7 Upper powerdomain is bifinite

lemma *approx-chain-upper-map*:

assumes *approx-chain* a
shows *approx-chain* $(\lambda i. upper-map.(a\ i))$
<proof>

instance *upper-pd* :: $(bifinite)\ bifinite$

<proof>

30.8 Join

definition

$upper-join :: 'a\ upper-pd\ upper-pd \rightarrow 'a\ upper-pd$ **where**
 $upper-join = (\Lambda\ xss. upper-bind.xss.(\Lambda\ xs. xs))$

lemma *upper-join-unit* [*simp*]:

$upper-join.\{xs\}\# = xs$
<proof>

lemma *upper-join-plus* [*simp*]:

$upper-join.(xss\ \cup\# \ yss) = upper-join.xss\ \cup\# \ upper-join.yss$
<proof>

lemma *upper-join-bottom* [*simp*]: $upper-join.\perp = \perp$

<proof>

lemma *upper-join-map-unit*:

$$\text{upper-join} \cdot (\text{upper-map} \cdot \text{upper-unit} \cdot xs) = xs$$

<proof>

lemma *upper-join-map-join*:

$$\text{upper-join} \cdot (\text{upper-map} \cdot \text{upper-join} \cdot xsss) = \text{upper-join} \cdot (\text{upper-join} \cdot xsss)$$

<proof>

lemma *upper-join-map-map*:

$$\text{upper-join} \cdot (\text{upper-map} \cdot (\text{upper-map} \cdot f) \cdot xss) = \\ \text{upper-map} \cdot f \cdot (\text{upper-join} \cdot xss)$$

<proof>

end

31 Lower powerdomain

theory *LowerPD*

imports *Compact-Basis*

begin

31.1 Basis preorder

definition

lower-le :: 'a pd-basis \Rightarrow 'a pd-basis \Rightarrow bool (**infix** \leq_b 50) **where**
lower-le = ($\lambda u v. \forall x \in \text{Rep-pd-basis } u. \exists y \in \text{Rep-pd-basis } v. x \sqsubseteq y$)

lemma *lower-le-refl* [*simp*]: $t \leq_b t$

<proof>

lemma *lower-le-trans*: $\llbracket t \leq_b u; u \leq_b v \rrbracket \Longrightarrow t \leq_b v$

<proof>

interpretation *lower-le*: preorder *lower-le*

<proof>

lemma *lower-le-minimal* [*simp*]: *PDUnit compact-bot* $\leq_b t$

<proof>

lemma *PDUnit-lower-mono*: $x \sqsubseteq y \Longrightarrow \text{PDUnit } x \leq_b \text{PDUnit } y$

<proof>

lemma *PDPlus-lower-mono*: $\llbracket s \leq_b t; u \leq_b v \rrbracket \Longrightarrow \text{PDPlus } s \ u \leq_b \text{PDPlus } t \ v$

<proof>

lemma *PDPlus-lower-le*: $t \leq_b \text{PDPlus } t \ u$

<proof>

lemma *lower-le-PDUnit-PDUnit-iff* [*simp*]:

$$(PDUnit\ a \leq_b PDUnit\ b) = (a \sqsubseteq b)$$

<proof>

lemma *lower-le-PDUnit-PDPlus-iff*:

$$(PDUnit\ a \leq_b PDPlus\ t\ u) = (PDUnit\ a \leq_b t \vee PDUnit\ a \leq_b u)$$

<proof>

lemma *lower-le-PDPlus-iff*: $(PDPlus\ t\ u \leq_b v) = (t \leq_b v \wedge u \leq_b v)$

<proof>

lemma *lower-le-induct* [*induct set: lower-le*]:

assumes *le*: $t \leq_b u$

assumes 1: $\bigwedge a\ b. a \sqsubseteq b \implies P\ (PDUnit\ a)\ (PDUnit\ b)$

assumes 2: $\bigwedge t\ u\ a. P\ (PDUnit\ a)\ t \implies P\ (PDUnit\ a)\ (PDPlus\ t\ u)$

assumes 3: $\bigwedge t\ u\ v. [P\ t\ v; P\ u\ v] \implies P\ (PDPlus\ t\ u)\ v$

shows $P\ t\ u$

<proof>

31.2 Type definition

typedef *'a lower-pd* $(((-)b)) =$

$\{S :: 'a\ pd\ basis\ set.\ lower\ le.\ ideal\ S\}$

<proof>

instantiation *lower-pd* :: (*bifinite*) *below*

begin

definition

$$x \sqsubseteq y \longleftrightarrow Rep\ lower\ pd\ x \subseteq Rep\ lower\ pd\ y$$

instance *<proof>*

end

instance *lower-pd* :: (*bifinite*) *po*

<proof>

instance *lower-pd* :: (*bifinite*) *cpo*

<proof>

definition

lower-principal :: *'a pd-basis* \implies *'a lower-pd* **where**

lower-principal $t = Abs\ lower\ pd\ \{u. u \leq_b t\}$

interpretation *lower-pd*:

ideal-completion lower-le lower-principal Rep-lower-pd

<proof>

Lower powerdomain is pointed

lemma *lower-pd-minimal*: *lower-principal* (PDUnit compact-bot) \sqsubseteq *ys*
 ⟨proof⟩

instance *lower-pd* :: (bifinite) *pcpo*
 ⟨proof⟩

lemma *inst-lower-pd-pcpo*: $\perp = \text{lower-principal}$ (PDUnit compact-bot)
 ⟨proof⟩

31.3 Monadic unit and plus

definition

lower-unit :: 'a \rightarrow 'a *lower-pd* **where**
lower-unit = compact-basis.extension ($\lambda a.$ *lower-principal* (PDUnit a))

definition

lower-plus :: 'a *lower-pd* \rightarrow 'a *lower-pd* \rightarrow 'a *lower-pd* **where**
lower-plus = *lower-pd.extension* ($\lambda t.$ *lower-pd.extension* ($\lambda u.$
lower-principal (PDPlus t u)))

abbreviation

lower-add :: 'a *lower-pd* \Rightarrow 'a *lower-pd* \Rightarrow 'a *lower-pd*
 (infixl \cup 65) **where**
xs \cup *ys* == *lower-plus*.*xs*.*ys*

syntax

-lower-pd :: *args* \Rightarrow *logic* ({-}b)

translations

{*x*,*xs*}b == {*x*}b \cup {*xs*}b
 {*x*}b == CONST *lower-unit*.*x*

lemma *lower-unit-Rep-compact-basis* [simp]:
 {*Rep-compact-basis* a}b = *lower-principal* (PDUnit a)
 ⟨proof⟩

lemma *lower-plus-principal* [simp]:
lower-principal t \cup *lower-principal* u = *lower-principal* (PDPlus t u)
 ⟨proof⟩

interpretation *lower-add*: *semilattice* *lower-add* ⟨proof⟩

lemmas *lower-plus-assoc* = *lower-add.assoc*

lemmas *lower-plus-commute* = *lower-add.commute*

lemmas *lower-plus-absorb* = *lower-add.idem*

lemmas *lower-plus-left-commute* = *lower-add.left-commute*

lemmas *lower-plus-left-absorb* = *lower-add.left-idem*

Useful for *simp add: lower-plus-ac*

lemmas *lower-plus-ac* =
lower-plus-assoc lower-plus-commute lower-plus-left-commute

Useful for *simp* only: *lower-plus-aci*

lemmas *lower-plus-aci* =
lower-plus-ac lower-plus-absorb lower-plus-left-absorb

lemma *lower-plus-below1*: $xs \sqsubseteq xs \cupb ys$
 ⟨*proof*⟩

lemma *lower-plus-below2*: $ys \sqsubseteq xs \cupb ys$
 ⟨*proof*⟩

lemma *lower-plus-least*: $\llbracket xs \sqsubseteq zs; ys \sqsubseteq zs \rrbracket \implies xs \cupb ys \sqsubseteq zs$
 ⟨*proof*⟩

lemma *lower-plus-below-iff* [*simp*]:
 $xs \cupb ys \sqsubseteq zs \longleftrightarrow xs \sqsubseteq zs \wedge ys \sqsubseteq zs$
 ⟨*proof*⟩

lemma *lower-unit-below-plus-iff* [*simp*]:
 $\{x\}b \sqsubseteq ys \cupb zs \longleftrightarrow \{x\}b \sqsubseteq ys \vee \{x\}b \sqsubseteq zs$
 ⟨*proof*⟩

lemma *lower-unit-below-iff* [*simp*]: $\{x\}b \sqsubseteq \{y\}b \longleftrightarrow x \sqsubseteq y$
 ⟨*proof*⟩

lemmas *lower-pd-below-simps* =
lower-unit-below-iff
lower-plus-below-iff
lower-unit-below-plus-iff

lemma *lower-unit-eq-iff* [*simp*]: $\{x\}b = \{y\}b \longleftrightarrow x = y$
 ⟨*proof*⟩

lemma *lower-unit-strict* [*simp*]: $\{\perp\}b = \perp$
 ⟨*proof*⟩

lemma *lower-unit-bottom-iff* [*simp*]: $\{x\}b = \perp \longleftrightarrow x = \perp$
 ⟨*proof*⟩

lemma *lower-plus-bottom-iff* [*simp*]:
 $xs \cupb ys = \perp \longleftrightarrow xs = \perp \wedge ys = \perp$
 ⟨*proof*⟩

lemma *lower-plus-strict1* [*simp*]: $\perp \cupb ys = ys$
 ⟨*proof*⟩

lemma *lower-plus-strict2* [*simp*]: $xs \cupb \perp = xs$

<proof>

lemma *compact-lower-unit*: $\text{compact } x \implies \text{compact } \{x\}^b$
<proof>

lemma *compact-lower-unit-iff* [simp]: $\text{compact } \{x\}^b \iff \text{compact } x$
<proof>

lemma *compact-lower-plus* [simp]:
 $\llbracket \text{compact } xs; \text{compact } ys \rrbracket \implies \text{compact } (xs \cup b \ ys)$
<proof>

31.4 Induction rules

lemma *lower-pd-induct1*:
assumes P : *adm* P
assumes *unit*: $\bigwedge x. P \ \{x\}^b$
assumes *insert*:
 $\bigwedge x \ ys. \llbracket P \ \{x\}^b; P \ ys \rrbracket \implies P \ (\{x\}^b \cup b \ ys)$
shows $P \ (xs::'a \ \text{lower-pd})$
<proof>

lemma *lower-pd-induct*
[*case-names adm lower-unit lower-plus, induct type: lower-pd*]:
assumes P : *adm* P
assumes *unit*: $\bigwedge x. P \ \{x\}^b$
assumes *plus*: $\bigwedge xs \ ys. \llbracket P \ xs; P \ ys \rrbracket \implies P \ (xs \cup b \ ys)$
shows $P \ (xs::'a \ \text{lower-pd})$
<proof>

31.5 Monadic bind

definition
lower-bind-basis ::
 $'a \ \text{pd-basis} \Rightarrow ('a \rightarrow 'b \ \text{lower-pd}) \rightarrow 'b \ \text{lower-pd}$ **where**
lower-bind-basis = *fold-pd*
 $(\lambda a. \Lambda f. f \cdot (\text{Rep-compact-basis } a))$
 $(\lambda x \ y. \Lambda f. x \cdot f \cup b \ y \cdot f)$

lemma *ACI-lower-bind*:
semilattice $(\lambda x \ y. \Lambda f. x \cdot f \cup b \ y \cdot f)$
<proof>

lemma *lower-bind-basis-simps* [simp]:
lower-bind-basis (*PDUnit* a) =
 $(\Lambda f. f \cdot (\text{Rep-compact-basis } a))$
lower-bind-basis (*PDPlus* $t \ u$) =
 $(\Lambda f. \text{lower-bind-basis } t \cdot f \cup b \ \text{lower-bind-basis } u \cdot f)$
<proof>

lemma *lower-bind-basis-mono*:

$t \leq_b u \implies \text{lower-bind-basis } t \sqsubseteq \text{lower-bind-basis } u$
 ⟨proof⟩

definition

$\text{lower-bind} :: 'a \text{ lower-pd} \rightarrow ('a \rightarrow 'b \text{ lower-pd}) \rightarrow 'b \text{ lower-pd}$ **where**
 $\text{lower-bind} = \text{lower-pd.extension lower-bind-basis}$

syntax

$\text{-lower-bind} :: [\text{logic}, \text{logic}, \text{logic}] \Rightarrow \text{logic}$
 $((\exists \cup_b \in \cdot / \cdot) [0, 0, 10] 10)$

translations

$\cup_b x \in xs. e == \text{CONST lower-bind} \cdot xs \cdot (\Lambda x. e)$

lemma *lower-bind-principal* [simp]:

$\text{lower-bind} \cdot (\text{lower-principal } t) = \text{lower-bind-basis } t$
 ⟨proof⟩

lemma *lower-bind-unit* [simp]:

$\text{lower-bind} \cdot \{x\}_b \cdot f = f \cdot x$
 ⟨proof⟩

lemma *lower-bind-plus* [simp]:

$\text{lower-bind} \cdot (xs \cup_b ys) \cdot f = \text{lower-bind} \cdot xs \cdot f \cup_b \text{lower-bind} \cdot ys \cdot f$
 ⟨proof⟩

lemma *lower-bind-strict* [simp]: $\text{lower-bind} \cdot \perp \cdot f = f \cdot \perp$

⟨proof⟩

lemma *lower-bind-bind*:

$\text{lower-bind} \cdot (\text{lower-bind} \cdot xs \cdot f) \cdot g = \text{lower-bind} \cdot xs \cdot (\Lambda x. \text{lower-bind} \cdot (f \cdot x) \cdot g)$
 ⟨proof⟩

31.6 Map

definition

$\text{lower-map} :: ('a \rightarrow 'b) \rightarrow 'a \text{ lower-pd} \rightarrow 'b \text{ lower-pd}$ **where**
 $\text{lower-map} = (\Lambda f xs. \text{lower-bind} \cdot xs \cdot (\Lambda x. \{f \cdot x\}_b))$

lemma *lower-map-unit* [simp]:

$\text{lower-map} \cdot f \cdot \{x\}_b = \{f \cdot x\}_b$
 ⟨proof⟩

lemma *lower-map-plus* [simp]:

$\text{lower-map} \cdot f \cdot (xs \cup_b ys) = \text{lower-map} \cdot f \cdot xs \cup_b \text{lower-map} \cdot f \cdot ys$
 ⟨proof⟩

lemma *lower-map-bottom* [simp]: $\text{lower-map} \cdot f \cdot \perp = \{f \cdot \perp\}_b$

<proof>

lemma *lower-map-ident*: $\text{lower-map} \cdot (\Lambda x. x) \cdot xs = xs$
<proof>

lemma *lower-map-ID*: $\text{lower-map} \cdot ID = ID$
<proof>

lemma *lower-map-map*:
 $\text{lower-map} \cdot f \cdot (\text{lower-map} \cdot g \cdot xs) = \text{lower-map} \cdot (\Lambda x. f \cdot (g \cdot x)) \cdot xs$
<proof>

lemma *lower-bind-map*:
 $\text{lower-bind} \cdot (\text{lower-map} \cdot f \cdot xs) \cdot g = \text{lower-bind} \cdot xs \cdot (\Lambda x. g \cdot (f \cdot x))$
<proof>

lemma *lower-map-bind*:
 $\text{lower-map} \cdot f \cdot (\text{lower-bind} \cdot xs \cdot g) = \text{lower-bind} \cdot xs \cdot (\Lambda x. \text{lower-map} \cdot f \cdot (g \cdot x))$
<proof>

lemma *ep-pair-lower-map*: $\text{ep-pair } e \text{ } p \implies \text{ep-pair } (\text{lower-map} \cdot e) (\text{lower-map} \cdot p)$
<proof>

lemma *deflation-lower-map*: $\text{deflation } d \implies \text{deflation } (\text{lower-map} \cdot d)$
<proof>

lemma *finite-deflation-lower-map*:
assumes *finite-deflation* *d* **shows** *finite-deflation* $(\text{lower-map} \cdot d)$
<proof>

31.7 Lower powerdomain is bifinite

lemma *approx-chain-lower-map*:
assumes *approx-chain* *a*
shows *approx-chain* $(\lambda i. \text{lower-map} \cdot (a \ i))$
<proof>

instance *lower-pd* :: $(\text{bifinite}) \text{ bifinite}$
<proof>

31.8 Join

definition
 $\text{lower-join} :: 'a \text{ lower-pd } \text{ lower-pd} \rightarrow 'a \text{ lower-pd}$ **where**
 $\text{lower-join} = (\Lambda xss. \text{lower-bind} \cdot xss \cdot (\Lambda xs. xs))$

lemma *lower-join-unit* [*simp*]:
 $\text{lower-join} \cdot \{xs\}^b = xs$
<proof>

lemma *lower-join-plus* [simp]:
 $lower-join.(xss \cupb yss) = lower-join.xss \cupb lower-join.yss$
 ⟨proof⟩

lemma *lower-join-bottom* [simp]: $lower-join.\perp = \perp$
 ⟨proof⟩

lemma *lower-join-map-unit*:
 $lower-join.(lower-map.lower-unit.xs) = xs$
 ⟨proof⟩

lemma *lower-join-map-join*:
 $lower-join.(lower-map.lower-join.xsss) = lower-join.(lower-join.xsss)$
 ⟨proof⟩

lemma *lower-join-map-map*:
 $lower-join.(lower-map.(lower-map.f).xss) =$
 $lower-map.f.(lower-join.xss)$
 ⟨proof⟩

end

32 Convex powerdomain

theory *ConvexPD*
imports *UpperPD LowerPD*
begin

32.1 Basis preorder

definition
 $convex-le :: 'a\ pd-basis \Rightarrow 'a\ pd-basis \Rightarrow bool$ (**infix** \leq_{\natural} 50) **where**
 $convex-le = (\lambda u\ v. u \leq_{\#} v \wedge u \leq_{\flat} v)$

lemma *convex-le-refl* [simp]: $t \leq_{\natural} t$
 ⟨proof⟩

lemma *convex-le-trans*: $\llbracket t \leq_{\natural} u; u \leq_{\natural} v \rrbracket \Longrightarrow t \leq_{\natural} v$
 ⟨proof⟩

interpretation *convex-le*: *preorder convex-le*
 ⟨proof⟩

lemma *upper-le-minimal* [simp]: $PDUnit\ compact-bot \leq_{\natural} t$
 ⟨proof⟩

lemma *PDUnit-convex-mono*: $x \sqsubseteq y \Longrightarrow PDUnit\ x \leq_{\natural} PDUnit\ y$
 ⟨proof⟩

lemma *PDPlus-convex-mono*: $\llbracket s \leq_{\sqsubseteq} t; u \leq_{\sqsubseteq} v \rrbracket \implies PDPlus\ s\ u \leq_{\sqsubseteq} PDPlus\ t\ v$
 ⟨proof⟩

lemma *convex-le-PDUnit-PDUnit-iff* [simp]:
 $(PDUnit\ a \leq_{\sqsubseteq} PDUnit\ b) = (a \sqsubseteq b)$
 ⟨proof⟩

lemma *convex-le-PDUnit-lemma1*:
 $(PDUnit\ a \leq_{\sqsubseteq} t) = (\forall b \in Rep\text{-}pd\text{-}basis\ t. a \sqsubseteq b)$
 ⟨proof⟩

lemma *convex-le-PDUnit-PDPlus-iff* [simp]:
 $(PDUnit\ a \leq_{\sqsubseteq} PDPlus\ t\ u) = (PDUnit\ a \leq_{\sqsubseteq} t \wedge PDUnit\ a \leq_{\sqsubseteq} u)$
 ⟨proof⟩

lemma *convex-le-PDUnit-lemma2*:
 $(t \leq_{\sqsubseteq} PDUnit\ b) = (\forall a \in Rep\text{-}pd\text{-}basis\ t. a \sqsubseteq b)$
 ⟨proof⟩

lemma *convex-le-PDPlus-PDUnit-iff* [simp]:
 $(PDPlus\ t\ u \leq_{\sqsubseteq} PDUnit\ a) = (t \leq_{\sqsubseteq} PDUnit\ a \wedge u \leq_{\sqsubseteq} PDUnit\ a)$
 ⟨proof⟩

lemma *convex-le-PDPlus-lemma*:
assumes $z: PDPlus\ t\ u \leq_{\sqsubseteq} z$
shows $\exists v\ w. z = PDPlus\ v\ w \wedge t \leq_{\sqsubseteq} v \wedge u \leq_{\sqsubseteq} w$
 ⟨proof⟩

lemma *convex-le-induct* [induct set: convex-le]:
assumes $le: t \leq_{\sqsubseteq} u$
assumes $2: \bigwedge t\ u\ v. \llbracket P\ t\ u; P\ u\ v \rrbracket \implies P\ t\ v$
assumes $3: \bigwedge a\ b. a \sqsubseteq b \implies P\ (PDUnit\ a)\ (PDUnit\ b)$
assumes $4: \bigwedge t\ u\ v\ w. \llbracket P\ t\ v; P\ u\ w \rrbracket \implies P\ (PDPlus\ t\ u)\ (PDPlus\ v\ w)$
shows $P\ t\ u$
 ⟨proof⟩

32.2 Type definition

typedef *'a convex-pd* $(('(-)\sqsubseteq)) =$
 $\{S :: 'a\ pd\text{-}basis\ set. convex\text{-}le.\ ideal\ S\}$
 ⟨proof⟩

instantiation *convex-pd* :: (bifinite) below
begin

definition
 $x \sqsubseteq y \longleftrightarrow Rep\text{-}convex\text{-}pd\ x \subseteq Rep\text{-}convex\text{-}pd\ y$

instance $\langle proof \rangle$
end

instance *convex-pd* :: (bifinite) po
 $\langle proof \rangle$

instance *convex-pd* :: (bifinite) cpo
 $\langle proof \rangle$

definition

convex-principal :: 'a pd-basis \Rightarrow 'a *convex-pd* **where**
convex-principal t = Abs-convex-pd {u. u \leq_{h} t}

interpretation *convex-pd*:

ideal-completion convex-le convex-principal Rep-convex-pd
 $\langle proof \rangle$

Convex powerdomain is pointed

lemma *convex-pd-minimal*: *convex-principal* (PDUnit compact-bot) \sqsubseteq ys
 $\langle proof \rangle$

instance *convex-pd* :: (bifinite) pcpo
 $\langle proof \rangle$

lemma *inst-convex-pd-pcpo*: $\perp = \text{convex-principal}$ (PDUnit compact-bot)
 $\langle proof \rangle$

32.3 Monadic unit and plus

definition

convex-unit :: 'a \rightarrow 'a *convex-pd* **where**
convex-unit = compact-basis.extension ($\lambda a.$ *convex-principal* (PDUnit a))

definition

convex-plus :: 'a *convex-pd* \rightarrow 'a *convex-pd* \rightarrow 'a *convex-pd* **where**
convex-plus = *convex-pd.extension* ($\lambda t.$ *convex-pd.extension* ($\lambda u.$
convex-principal (PDPlus t u)))

abbreviation

convex-add :: 'a *convex-pd* \Rightarrow 'a *convex-pd* \Rightarrow 'a *convex-pd*
(infixl \cup_{h} 65) **where**
 $xs \cup_{\text{h}} ys == \text{convex-plus} \cdot xs \cdot ys$

syntax

-convex-pd :: args \Rightarrow logic ($\{-\}_{\text{h}}$)

translations

$\{x, xs\}_{\text{h}} == \{x\}_{\text{h}} \cup_{\text{h}} \{xs\}_{\text{h}}$
 $\{x\}_{\text{h}} == \text{CONST } \text{convex-unit} \cdot x$

lemma *convex-unit-Rep-compact-basis* [simp]:
 $\{\text{Rep-compact-basis } a\} \sqsubseteq = \text{convex-principal } (\text{PDUnit } a)$
 ⟨proof⟩

lemma *convex-plus-principal* [simp]:
 $\text{convex-principal } t \cup \sqsubseteq \text{convex-principal } u = \text{convex-principal } (\text{PDPlus } t \ u)$
 ⟨proof⟩

interpretation *convex-add: semilattice convex-add* ⟨proof⟩

lemmas *convex-plus-assoc = convex-add.assoc*
lemmas *convex-plus-commute = convex-add.commute*
lemmas *convex-plus-absorb = convex-add.idem*
lemmas *convex-plus-left-commute = convex-add.left-commute*
lemmas *convex-plus-left-absorb = convex-add.left-idem*

Useful for *simp add: convex-plus-ac*

lemmas *convex-plus-ac =*
convex-plus-assoc convex-plus-commute convex-plus-left-commute

Useful for *simp only: convex-plus-aci*

lemmas *convex-plus-aci =*
convex-plus-ac convex-plus-absorb convex-plus-left-absorb

lemma *convex-unit-below-plus-iff* [simp]:
 $\{x\} \sqsubseteq ys \cup \sqsubseteq zs \longleftrightarrow \{x\} \sqsubseteq ys \wedge \{x\} \sqsubseteq zs$
 ⟨proof⟩

lemma *convex-plus-below-unit-iff* [simp]:
 $xs \cup \sqsubseteq ys \sqsubseteq \{z\} \longleftrightarrow xs \sqsubseteq \{z\} \wedge ys \sqsubseteq \{z\}$
 ⟨proof⟩

lemma *convex-unit-below-iff* [simp]: $\{x\} \sqsubseteq \{y\} \longleftrightarrow x \sqsubseteq y$
 ⟨proof⟩

lemma *convex-unit-eq-iff* [simp]: $\{x\} = \{y\} \longleftrightarrow x = y$
 ⟨proof⟩

lemma *convex-unit-strict* [simp]: $\{\perp\} = \perp$
 ⟨proof⟩

lemma *convex-unit-bottom-iff* [simp]: $\{x\} = \perp \longleftrightarrow x = \perp$
 ⟨proof⟩

lemma *compact-convex-unit*: $\text{compact } x \implies \text{compact } \{x\}$
 ⟨proof⟩

lemma *compact-convex-unit-iff* [simp]: $\text{compact } \{x\} \longleftrightarrow \text{compact } x$

<proof>

lemma *compact-convex-plus* [*simp*]:

$\llbracket \text{compact } xs; \text{ compact } ys \rrbracket \implies \text{compact } (xs \cup\!\!\!\sqcup ys)$

<proof>

32.4 Induction rules

lemma *convex-pd-induct1*:

assumes $P: \text{adm } P$

assumes *unit*: $\bigwedge x. P \{x\}$

assumes *insert*: $\bigwedge x \text{ ys}. \llbracket P \{x\}; P \text{ ys} \rrbracket \implies P (\{x\} \cup\!\!\!\sqcup \text{ys})$

shows $P (xs::'a \text{ convex-pd})$

<proof>

lemma *convex-pd-induct*

[*case-names adm convex-unit convex-plus, induct type: convex-pd*]:

assumes $P: \text{adm } P$

assumes *unit*: $\bigwedge x. P \{x\}$

assumes *plus*: $\bigwedge xs \text{ ys}. \llbracket P xs; P ys \rrbracket \implies P (xs \cup\!\!\!\sqcup ys)$

shows $P (xs::'a \text{ convex-pd})$

<proof>

32.5 Monadic bind

definition

convex-bind-basis ::

$'a \text{ pd-basis} \implies ('a \rightarrow 'b \text{ convex-pd}) \rightarrow 'b \text{ convex-pd}$ **where**

convex-bind-basis = *fold-pd*

($\lambda a. \Lambda f. f \cdot (\text{Rep-compact-basis } a)$)

($\lambda x y. \Lambda f. x \cdot f \cup\!\!\!\sqcup y \cdot f$)

lemma *ACI-convex-bind*:

semilattice ($\lambda x y. \Lambda f. x \cdot f \cup\!\!\!\sqcup y \cdot f$)

<proof>

lemma *convex-bind-basis-simps* [*simp*]:

convex-bind-basis (*PDUnit* a) =

($\Lambda f. f \cdot (\text{Rep-compact-basis } a)$)

convex-bind-basis (*PDPlus* $t \ u$) =

($\Lambda f. \text{convex-bind-basis } t \cdot f \cup\!\!\!\sqcup \text{convex-bind-basis } u \cdot f$)

<proof>

lemma *convex-bind-basis-mono*:

$t \leq\!\!\!\sqcup u \implies \text{convex-bind-basis } t \sqsubseteq \text{convex-bind-basis } u$

<proof>

definition

convex-bind :: $'a \text{ convex-pd} \rightarrow ('a \rightarrow 'b \text{ convex-pd}) \rightarrow 'b \text{ convex-pd}$ **where**

convex-bind = *convex-pd.extension convex-bind-basis*

syntax

-convex-bind :: [logic, logic, logic] ⇒ logic
 ((∃ ∪ ∓ ∈ - / -) [0, 0, 10] 10)

translations

$\bigcup \{x \in xs. e\} == \text{CONST } \text{convex-bind} \cdot xs \cdot (\Lambda x. e)$

lemma convex-bind-principal [simp]:

$\text{convex-bind} \cdot (\text{convex-principal } t) = \text{convex-bind-basis } t$
 ⟨proof⟩

lemma convex-bind-unit [simp]:

$\text{convex-bind} \cdot \{x\} \sqsupset f = f \cdot x$
 ⟨proof⟩

lemma convex-bind-plus [simp]:

$\text{convex-bind} \cdot (xs \cup \sqsupset ys) \cdot f = \text{convex-bind} \cdot xs \cdot f \cup \sqsupset \text{convex-bind} \cdot ys \cdot f$
 ⟨proof⟩

lemma convex-bind-strict [simp]: $\text{convex-bind} \cdot \perp \cdot f = f \cdot \perp$

⟨proof⟩

lemma convex-bind-bind:

$\text{convex-bind} \cdot (\text{convex-bind} \cdot xs \cdot f) \cdot g =$
 $\text{convex-bind} \cdot xs \cdot (\Lambda x. \text{convex-bind} \cdot (f \cdot x) \cdot g)$
 ⟨proof⟩

32.6 Map

definition

$\text{convex-map} :: ('a \rightarrow 'b) \rightarrow 'a \text{ convex-pd} \rightarrow 'b \text{ convex-pd}$ **where**
 $\text{convex-map} = (\Lambda f \ xs. \text{convex-bind} \cdot xs \cdot (\Lambda x. \{f \cdot x\} \sqsupset))$

lemma convex-map-unit [simp]:

$\text{convex-map} \cdot f \cdot \{x\} \sqsupset = \{f \cdot x\} \sqsupset$
 ⟨proof⟩

lemma convex-map-plus [simp]:

$\text{convex-map} \cdot f \cdot (xs \cup \sqsupset ys) = \text{convex-map} \cdot f \cdot xs \cup \sqsupset \text{convex-map} \cdot f \cdot ys$
 ⟨proof⟩

lemma convex-map-bottom [simp]: $\text{convex-map} \cdot f \cdot \perp = \{f \cdot \perp\} \sqsupset$

⟨proof⟩

lemma convex-map-ident: $\text{convex-map} \cdot (\Lambda x. x) \cdot xs = xs$

⟨proof⟩

lemma convex-map-ID: $\text{convex-map} \cdot \text{ID} = \text{ID}$

<proof>

lemma *convex-map-map*:

$$\text{convex-map}\cdot f\cdot(\text{convex-map}\cdot g\cdot xs) = \text{convex-map}\cdot(\Lambda x. f\cdot(g\cdot x))\cdot xs$$

<proof>

lemma *convex-bind-map*:

$$\text{convex-bind}\cdot(\text{convex-map}\cdot f\cdot xs)\cdot g = \text{convex-bind}\cdot xs\cdot(\Lambda x. g\cdot(f\cdot x))$$

<proof>

lemma *convex-map-bind*:

$$\text{convex-map}\cdot f\cdot(\text{convex-bind}\cdot xs\cdot g) = \text{convex-bind}\cdot xs\cdot(\Lambda x. \text{convex-map}\cdot f\cdot(g\cdot x))$$

<proof>

lemma *ep-pair-convex-map*: $ep\text{-pair } e\ p \implies ep\text{-pair } (\text{convex-map}\cdot e)\ (\text{convex-map}\cdot p)$

<proof>

lemma *deflation-convex-map*: $deflation\ d \implies deflation\ (\text{convex-map}\cdot d)$

<proof>

lemma *finite-deflation-convex-map*:

assumes *finite-deflation* d **shows** *finite-deflation* $(\text{convex-map}\cdot d)$

<proof>

32.7 Convex powerdomain is bifinite

lemma *approx-chain-convex-map*:

assumes *approx-chain* a

shows *approx-chain* $(\lambda i. \text{convex-map}\cdot(a\ i))$

<proof>

instance *convex-pd* :: $(bifinite)\ bifinite$

<proof>

32.8 Join

definition

convex-join :: $'a\ \text{convex-pd}\ \text{convex-pd} \rightarrow 'a\ \text{convex-pd}$ **where**

$$\text{convex-join} = (\Lambda\ xss. \text{convex-bind}\cdot xss\cdot(\Lambda\ xs. xs))$$

lemma *convex-join-unit* [*simp*]:

$$\text{convex-join}\cdot\{xs\}\natural = xs$$

<proof>

lemma *convex-join-plus* [*simp*]:

$$\text{convex-join}\cdot(xss\ \cup\!\!\!\cup\ yss) = \text{convex-join}\cdot xss\ \cup\!\!\!\cup\ \text{convex-join}\cdot yss$$

<proof>

lemma *convex-join-bottom* [*simp*]: $\text{convex-join}\cdot\perp = \perp$

<proof>

lemma *convex-join-map-unit:*

$$\text{convex-join} \cdot (\text{convex-map} \cdot \text{convex-unit} \cdot xs) = xs$$

<proof>

lemma *convex-join-map-join:*

$$\text{convex-join} \cdot (\text{convex-map} \cdot \text{convex-join} \cdot xsss) = \text{convex-join} \cdot (\text{convex-join} \cdot xsss)$$

<proof>

lemma *convex-join-map-map:*

$$\begin{aligned} \text{convex-join} \cdot (\text{convex-map} \cdot (\text{convex-map} \cdot f) \cdot xss) = \\ \text{convex-map} \cdot f \cdot (\text{convex-join} \cdot xss) \end{aligned}$$

<proof>

32.9 Conversions to other powerdomains

Convex to upper

lemma *convex-le-imp-upper-le:* $t \leq_{\natural} u \implies t \leq_{\sharp} u$

<proof>

definition

convex-to-upper :: 'a convex-pd \rightarrow 'a upper-pd **where**
convex-to-upper = *convex-pd.extension upper-principal*

lemma *convex-to-upper-principal [simp]:*

$$\text{convex-to-upper} \cdot (\text{convex-principal } t) = \text{upper-principal } t$$

<proof>

lemma *convex-to-upper-unit [simp]:*

$$\text{convex-to-upper} \cdot \{x\}_{\natural} = \{x\}_{\sharp}$$

<proof>

lemma *convex-to-upper-plus [simp]:*

$$\text{convex-to-upper} \cdot (xs \cup_{\natural} ys) = \text{convex-to-upper} \cdot xs \cup_{\sharp} \text{convex-to-upper} \cdot ys$$

<proof>

lemma *convex-to-upper-bind [simp]:*

$$\begin{aligned} \text{convex-to-upper} \cdot (\text{convex-bind} \cdot xs \cdot f) = \\ \text{upper-bind} \cdot (\text{convex-to-upper} \cdot xs) \cdot (\text{convex-to-upper} \text{ oo } f) \end{aligned}$$

<proof>

lemma *convex-to-upper-map [simp]:*

$$\text{convex-to-upper} \cdot (\text{convex-map} \cdot f \cdot xs) = \text{upper-map} \cdot f \cdot (\text{convex-to-upper} \cdot xs)$$

<proof>

lemma *convex-to-upper-join [simp]:*

$$\begin{aligned} \text{convex-to-upper} \cdot (\text{convex-join} \cdot xss) = \\ \text{upper-bind} \cdot (\text{convex-to-upper} \cdot xss) \cdot \text{convex-to-upper} \end{aligned}$$

<proof>

Convex to lower

lemma *convex-le-imp-lower-le*: $t \leq^{\mathfrak{a}} u \implies t \leq^{\mathfrak{b}} u$

<proof>

definition

convex-to-lower :: 'a convex-pd \rightarrow 'a lower-pd **where**
convex-to-lower = *convex-pd.extension lower-principal*

lemma *convex-to-lower-principal* [*simp*]:

convex-to-lower.(*convex-principal* t) = *lower-principal* t

<proof>

lemma *convex-to-lower-unit* [*simp*]:

convex-to-lower.{x}[ⓐ] = {x}[ⓑ]

<proof>

lemma *convex-to-lower-plus* [*simp*]:

convex-to-lower.(xs $\cup^{\mathfrak{a}}$ ys) = *convex-to-lower*.xs $\cup^{\mathfrak{b}}$ *convex-to-lower*.ys

<proof>

lemma *convex-to-lower-bind* [*simp*]:

convex-to-lower.(*convex-bind*.xs.f) =
lower-bind.(*convex-to-lower*.xs).(convex-to-lower oo f)

<proof>

lemma *convex-to-lower-map* [*simp*]:

convex-to-lower.(*convex-map*.f.xs) = *lower-map*.f.(*convex-to-lower*.xs)

<proof>

lemma *convex-to-lower-join* [*simp*]:

convex-to-lower.(*convex-join*.xss) =
lower-bind.(*convex-to-lower*.xss).convex-to-lower

<proof>

Ordering property

lemma *convex-pd-below-iff*:

(xs \sqsubseteq ys) =
 (*convex-to-upper*.xs \sqsubseteq *convex-to-upper*.ys \wedge
convex-to-lower.xs \sqsubseteq *convex-to-lower*.ys)

<proof>

lemmas *convex-plus-below-plus-iff* =

convex-pd-below-iff [**where** xs=xs $\cup^{\mathfrak{a}}$ ys **and** ys=zs $\cup^{\mathfrak{a}}$ ws]
for xs ys zs ws

lemmas *convex-pd-below-simps* =

convex-unit-below-plus-iff


```

convex-plus-below-unit-iff
convex-plus-below-plus-iff
convex-unit-below-iff
convex-to-upper-unit
convex-to-upper-plus
convex-to-lower-unit
convex-to-lower-plus
upper-pd-below-simps
lower-pd-below-simps

```

end

33 Powerdomains

```

theory Powerdomains
imports ConvexPD Domain
begin

```

33.1 Universal domain embeddings

definition *upper-emb* = *udom-emb* ($\lambda i.$ *upper-map*·(*udom-approx* *i*))

definition *upper-prj* = *udom-prj* ($\lambda i.$ *upper-map*·(*udom-approx* *i*))

definition *lower-emb* = *udom-emb* ($\lambda i.$ *lower-map*·(*udom-approx* *i*))

definition *lower-prj* = *udom-prj* ($\lambda i.$ *lower-map*·(*udom-approx* *i*))

definition *convex-emb* = *udom-emb* ($\lambda i.$ *convex-map*·(*udom-approx* *i*))

definition *convex-prj* = *udom-prj* ($\lambda i.$ *convex-map*·(*udom-approx* *i*))

lemma *ep-pair-upper*: *ep-pair upper-emb upper-prj*
 ⟨*proof*⟩

lemma *ep-pair-lower*: *ep-pair lower-emb lower-prj*
 ⟨*proof*⟩

lemma *ep-pair-convex*: *ep-pair convex-emb convex-prj*
 ⟨*proof*⟩

33.2 Deflation combinators

definition *upper-defl* :: *udom defl* → *udom defl*
 where *upper-defl* = *defl-fun1 upper-emb upper-prj upper-map*

definition *lower-defl* :: *udom defl* → *udom defl*
 where *lower-defl* = *defl-fun1 lower-emb lower-prj lower-map*

definition *convex-defl* :: *udom defl* → *udom defl*
 where *convex-defl* = *defl-fun1 convex-emb convex-prj convex-map*

lemma *cast-upper-defl*:

$cast.(upper-defl.A) = upper-emb \text{ oo } upper-map.(cast.A) \text{ oo } upper-prj$
 $\langle proof \rangle$

lemma *cast-lower-defl*:

$cast.(lower-defl.A) = lower-emb \text{ oo } lower-map.(cast.A) \text{ oo } lower-prj$
 $\langle proof \rangle$

lemma *cast-convex-defl*:

$cast.(convex-defl.A) = convex-emb \text{ oo } convex-map.(cast.A) \text{ oo } convex-prj$
 $\langle proof \rangle$

33.3 Domain class instances

instantiation *upper-pd* :: (domain) domain

begin

definition

$emb = upper-emb \text{ oo } upper-map.emb$

definition

$prj = upper-map.prj \text{ oo } upper-prj$

definition

$defl (t::'a \text{ upper-pd } itself) = upper-defl.DEFL('a)$

definition

$(liftemb :: 'a \text{ upper-pd } u \rightarrow udom \ u) = u-map.emb$

definition

$(liftprj :: udom \ u \rightarrow 'a \text{ upper-pd } u) = u-map.prj$

definition

$liftdefl (t::'a \text{ upper-pd } itself) = liftdefl-of.DEFL('a \text{ upper-pd})$

instance $\langle proof \rangle$

end

instantiation *lower-pd* :: (domain) domain

begin

definition

$emb = lower-emb \text{ oo } lower-map.emb$

definition

$prj = lower-map.prj \text{ oo } lower-prj$

definition

$\text{defl } (t::'a \text{ lower-pd itself}) = \text{lower-defl} \cdot \text{DEFL}('a)$

definition

$(\text{liftemb} :: 'a \text{ lower-pd } u \rightarrow \text{udom } u) = u\text{-map} \cdot \text{emb}$

definition

$(\text{liftprj} :: \text{udom } u \rightarrow 'a \text{ lower-pd } u) = u\text{-map} \cdot \text{prj}$

definition

$\text{liftdefl } (t::'a \text{ lower-pd itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \text{ lower-pd})$

instance $\langle \text{proof} \rangle$

end

instantiation $\text{convex-pd} :: (\text{domain}) \text{ domain}$

begin

definition

$\text{emb} = \text{convex-emb} \text{ oo } \text{convex-map} \cdot \text{emb}$

definition

$\text{prj} = \text{convex-map} \cdot \text{prj} \text{ oo } \text{convex-prj}$

definition

$\text{defl } (t::'a \text{ convex-pd itself}) = \text{convex-defl} \cdot \text{DEFL}('a)$

definition

$(\text{liftemb} :: 'a \text{ convex-pd } u \rightarrow \text{udom } u) = u\text{-map} \cdot \text{emb}$

definition

$(\text{liftprj} :: \text{udom } u \rightarrow 'a \text{ convex-pd } u) = u\text{-map} \cdot \text{prj}$

definition

$\text{liftdefl } (t::'a \text{ convex-pd itself}) = \text{liftdefl-of} \cdot \text{DEFL}('a \text{ convex-pd})$

instance $\langle \text{proof} \rangle$

end

lemma $\text{DEFL-upper}: \text{DEFL}('a::\text{domain upper-pd}) = \text{upper-defl} \cdot \text{DEFL}('a)$

$\langle \text{proof} \rangle$

lemma $\text{DEFL-lower}: \text{DEFL}('a::\text{domain lower-pd}) = \text{lower-defl} \cdot \text{DEFL}('a)$

$\langle \text{proof} \rangle$

lemma $\text{DEFL-convex}: \text{DEFL}('a::\text{domain convex-pd}) = \text{convex-defl} \cdot \text{DEFL}('a)$

$\langle \text{proof} \rangle$

33.4 Isomorphic deflations

lemma *isodefl-upper:*

$isodefl\ d\ t \implies isodefl\ (upper-map \cdot d)\ (upper-defl \cdot t)$
<proof>

lemma *isodefl-lower:*

$isodefl\ d\ t \implies isodefl\ (lower-map \cdot d)\ (lower-defl \cdot t)$
<proof>

lemma *isodefl-convex:*

$isodefl\ d\ t \implies isodefl\ (convex-map \cdot d)\ (convex-defl \cdot t)$
<proof>

33.5 Domain package setup for powerdomains

lemmas [*domain-defl-simps*] = *DEFL-upper DEFL-lower DEFL-convex*

lemmas [*domain-map-ID*] = *upper-map-ID lower-map-ID convex-map-ID*

lemmas [*domain-isodefl*] = *isodefl-upper isodefl-lower isodefl-convex*

lemmas [*domain-deflation*] =

deflation-upper-map deflation-lower-map deflation-convex-map

<ML>

end

theory *HOLCF*

imports

Main

Domain

Powerdomains

begin

default-sort *domain*

end